MASTER EN SCIENCES INFORMATIQUES

Startup - Configuration automatisée et dérivation de produit

Thiebaut, François

Award date:
2015

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2014–2015

Startup – Configuration automatisée et dérivation de produit

François Thiébaut

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Patrick Heymans

Co-promoteur : Maxime Cordy

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Avant-propos

J'aimerais remercier particulièrement mon co-promoteur Maxime Cordy pour son suivi tout au long du mémoire et pour ses nombreux conseils qui m'ont aidé à la rédaction de ce document.

Merci également à Patrick Heymans et Germain Saval pour l'aide apportée.

Table des matières

Table des figures

Introduction

Depuis l'émergence des lignes de produits logiciels, nous rencontrons de plus en plus d'outils et de logiciels permettant aux utilisateurs de choisir eux-mêmes les caractéristiques d'un produit pour qu'il réponde à leurs besoins, que ce soit pour une voiture, un ordinateur ou encore un vêtement.

Pour répondre à ce besoin de customisation, l'utilisateur se sert d'un outil appelé configurateur. Ce type de logiciel présente à l'utilisateur une liste d'options à choisir afin d'atteindre un résultat s'accordant à ses besoins. Des configurateurs peuvent être trouvés en ligne, typiquement sur des sites de ventes automobiles où l'on se voit proposer de créer soit même sa propre voiture via une interface offrant les différentes options possibles.

Les configurateurs actuels sont limités aux contextes dans lesquels ils sont développés. Ils sont créés uniquement afin de répondre à un but précis (comme par exemple, pour le configurateur de voiture en ligne). Face à une demande d'outils de configuration croissante et touchant plusieurs domaines, nous devons nous attendre à devoir gérer des espaces de possibilités plus conséquents, plus larges. Cela amène un niveau de complexité supérieur avec plus de contraintes à prendre en compte. Concevoir un configurateur comme un simple logiciel ou une simple application web n'est par conséquent plus approprié. Il est donc nécessaire de généraliser le processus et de le rendre accessible à plusieurs domaines.

C'est pour répondre à ce problème que le centre de recherche PReCISE a développé une série d'outils permettant de soulager les concepteurs de la production manuelle de code [5]. Cette série d'outils baptisée CONFETOOLS repose sur la séparation de la modélisation du domaine (contraintes,...) et de la partie logicielle (interface utilisateur,...). Concernant la première partie, un langage de modélisation a été mis au point et permet de spécifier une liste d'options, de sous options et de contraintes associées de façon compréhensible (*human-readable*), appelé TVL (Textual Variability Language) [6]. Quant à la partie IHM, la génération d'une interface se fait grâce à un autre langage permettant de produire des "vues" pour l'utilisateur depuis le modèle TVL, celui-ci est appelé TVDL. Un autre langage permettant de définir le design selon le contenu du modèle TVL a également été utilisé et est appelé FCSS (Featured Cascading Style Sheets). TVDL et FCSS sont donc utilisés pour définir le *look-and-feel* de l'interface graphique. Une fois ces parties définies, le configurateur est généré. L'utilisateur est dès lors libre d'effectuer ses choix parmi ce qui lui est proposé, lui permettant d'atteindre une configuration respectant ses besoins. Les contraintes sont gérées en temps réel, ce qui fait que lorsque l'utilisateur effectue un choix, les contraintes qui y sont liées ont des conséquences dans les options suivantes ; celles-ci peuvent par exemple réduire une liste de choix ou imposer une valeur. Ces conséquences

sont appelées des propagations.

Le fait d'avoir ces larges espaces de possibilités cause un problème de navigation. En effet, dans un contexte de configuration, un nombre important de possibilités et de combinaisons d'options possibles résultent en un nombre de configurations potentielles considérable. Cela rend la navigation complexe et chronophage si l'utilisateur souhaite en parcourir plusieurs. En effet, nous pouvons nous attendre à ce que l'utilisateur désire tester différentes configurations alternatives et de les comparer afin de choisir celle qui convient le mieux à ce qu'il cherche. Pour ce faire, il doit choisir différentes options à différents moments. Or, le configurateur ne permet pour le moment que de tester une configuration à la fois. Si l'utilisateur souhaite comparer différentes configurations, il est actuellement obligé d'annuler ce qu'il a réalisé et de recommencer son travail en choisissant d'autres options afin d'atteindre un résultat différent. Avec un tel nombre de possibilités, il est nécessaire que l'utilisateur puisse naviguer dans ses configurations de manière efficiente. Nous identifions ici ce qui va être le problème principal de ce mémoire : permettre à l'utilisateur de revenir en arrière dans son processus de configuration et lui permettre de comparer différentes configurations de manière aisée grâce à un système de navigation approprié.

Le retour arrière, plus communément appelé UNDO, est une fonction bien connue et utilisée dans presque tous les logiciels actuels. L'UNDO permet d'annuler les commandes effectuées par l'utilisateur et de retourner à un état antérieur du logiciel. Si nous prenons un logiciel d'édition de texte comme Microsoft Word, l'UNDO servira le plus souvent à annuler les dernières frappes, dernières insertions d'objets ou modification d'éléments. Cela permettra ainsi à l'utilisateur de corriger ses erreurs et d'effectuer de nouvelles commandes. Lorsque nous mentionnons l'UNDO nous nous devons de mentionner l'autre commande qui lui est associée : le REDO, qui sert à ré-exécuter les commandes précédemment défaites par l'UNDO.

L'application de l'UNDO à la configuration soulève certains problèmes que nous devrons prendre en compte lors de l'élaboration de notre solution. Premièrement, nous avons la gestion des contraintes impliquées par certains choix dans une configuration. Ces contraintes peuvent reprendre la suppression ou l'imposition de futurs choix. Si un UNDO est effectué, nous devons nous assurer que ces contraintes propagées sont également UNDO en même temps que la commande les ayant provoquées. De la même manière le REDO d'une commande ayant originellement provoqué des propagations doit de nouveau provoquer ces dernières. Un second problème est lié à la cohérence des états après chaque UNDO/REDO. Par exemple, si nous effectuons des UNDO sur deux choix successifs mais que le second dépend de l'existence du premier, alors il doit être impossible d'exécuter un REDO sur le second tant que le premier ne l'a pas lui-même été. Ces deux aspects sont importants à prendre en compte lors de l'élaboration d'un système d'UNDO/REDO dans la configuration.

Nous souhaitons dans ce mémoire appliquer un système d'UNDO/REDO au configurateur, combiné avec un système de navigation dans les configurations. Par combiné, nous entendons qu'il y aura d'un côté, une structure reprenant les séquences de configuration de l'utilisateur et de l'autre un historique des commandes exécutées pour obtenir la structure.

Cette dernière lui permettra d'obtenir une vue d'ensemble sur son travail, d'y naviguer aisément et d'effectuer des retours à des configurations antérieures. Cela lui permettra de tester d'autres choix et de les comparer avec d'autres configurations. L'historique de commande offrira une vue précise des commandes effectuées et permettra à l'utilisateur de les UNDO ou de les REDO.

Dans le premier chapitre de ce mémoire, identifierons et étudierons d'abord les différents travaux qui ont déjà été effectués sur l'UNDO et le REDO.

Le second chapitre sera consacré à l'élaboration de critères d'application d'un modèle d'UNDO au contexte de la configuration. Sur base de ces critères, nous évaluerons les modèles existants afin de savoir lesquels sont appropriés pour résoudre le problème. Les résultats de cette évaluation mettront en évidence qu'aucun modèle ne correspond entièrement à la configuration.

Pour cette raison, nous élaborerons notre propre solution dans le troisième chapitre sur base des qualités des modèles évalués. Cette solution sera un système permettant la navigation dans les différentes configurations créées par un utilisateur, combiné à un modèle d'UNDO/REDO.

Nous décrirons, dans le quatrième chapitre, l'implémentation du modèle que nous avons réalisé en utilisant les outils CONFETOOLS permettant l'utilisation des services de configuration.

L'ultime chapitre enchainera sur une évaluation du modèle développé en le soumettant à un scénario de configuration. Nous verrons que le nouveau modèle s'applique au scénario et au cadre de la configuration. Nous évaluerons ensuite ses avantages dans ce contexte par rapports aux autres modèles existants.

Nous conclurons en parlant des perspectives d'amélioration de notre solution.

Chapitre I

État de l'art

Le retour-arrière ou UNDO est une des fonctions les plus utilisées au monde, nous pouvons la retrouver dans presque tous les logiciels. Son effet est d'annuler l'exécution d'une action effectuée par l'utilisateur qui modifie l'état du programme, nous appelons une telle action une **commande**. Quand l'utilisateur procède à un UNDO sur une commande, cela rétablit le programme dans l'état dans lequel il était avant l'exécution de la commande. Une commande s'appliquant sur une autre est appelée **méta-commande**.
Nous ne pouvons pas parler de l'UNDO sans mentionner son contraire, le REDO. Si l'UNDO annule l'exécution d'une commande, le REDO rétablit ce qui a été annulé.
La fonction UNDO n'est pas toujours interprétée et utilisée de la même manière, ce qui offre plusieurs possibilités d'implémentation. Et si on prend en compte le REDO, le nombre de possibilités augmente encore. Beaucoup de chercheurs ont déjà apporté leurs pierres à l'édifice et en ont développé différentes versions. Nous allons donc, en premier lieu, découvrir quels travaux concernant l'UNDO et le REDO ont déjà été effectué.
Les différentes implémentations de ces fonctions sont appelés des modèles. Nous allons présenter ceux existants en les introduisant d'abord et ensuite en analysant leur capacité à s'appliquer dans un processus de configuration. Précisons que nous ne prendrons pas en compte les modèles d'UNDO visant des manipulations graphiques, celles-ci n'étant pas pertinentes dans un contexte de configuration.

I.1 Modèle linéaire

Le modèle linéaire se compose de deux listes : une *history list* et une *redo list*. La première contient toutes les commandes effectuées chronologiquement par l'utilisateur. La seconde contient les commandes qui ont déjà été défaites (UNDO) et que l'utilisateur peut ré-exécuter s'il le souhaite (REDO).

Représentons le modèle linéaire m comme un couple $m = (h, r)$ avec h, une liste de commande $(c_1, ..., c_m)$ appelée *history list* et avec r est une liste de commandes $(c_1, ..., c_n)$ appelée *redo list*. Initialement, ces deux listes sont vides. Nous appellerons DO, l'exécution d'une commande c par l'utilisateur. Le modèle linéaire définit l'application de DO sur m comme un couple (m, c) résultant en un couple (h', r) avec $h' = (c_1, ..., c_m, c)$. Dans ce modèle, c'est uniquement la dernière action dans l'*history list* qui peut être défaite. Dès que l'utilisateur procède à un UNDO, la dernière commande dans l'*history list* est supprimée et placée au début de la *redo list* (Voir Figure I.1). Dans cette situation, après

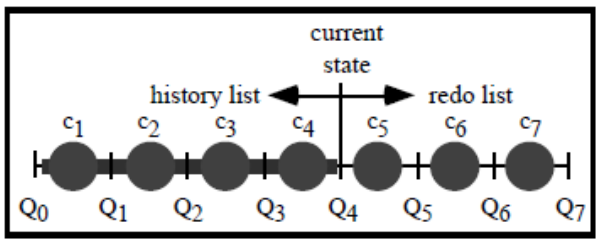


FIGURE I.1 – Illustration du modèle linéaire [1]

l'application de l'UNDO, définit comme UNDO(m), nous obtenons un couple (h', r') avec $h' = (c_1, .., c_{m-1})$ et $r' = (c_m, c'_1, ..., c'_n)$. Au contraire, lorsque l'utilisateur procède à un REDO, la première commande de la *redo list* est supprimée et ajoutée à la fin de l'*history list*. Nous avons REDO(m) $= (h', r')$ où $h' = (c_1, ..., c_m, c'_1)$ et où $r' = (c'_2, ..., c'_n)$.
Pour défaire uniquement une commande souhaitée (par exemple : c_2 dans la Fig. I.1), l'utilisateur doit d'abord effectuer un UNDO sur toutes les autres commandes accomplies depuis celle ciblée (c_3 et c_4 dans notre exemple). De la même manière, si un utilisateur souhaite REDO une commande précise, il doit exécuter un REDO sur toutes celles la précédant dans la redo list.

Définissons un exemple que nous utiliserons pour illustrer le fonctionnement de chacun de nos modèles.

Exemple 1. *Exemple de substitution : Prenons une liste h de commandes exécutées $(c_1, c_2, c_3, c_4, c_5)$. Le but est d'observer comment les méta-commandes d'un modèle peuvent être utilisées afin de substituer la commande c_2 par une commande c'_2 pour finalement obtenir $h = (c_1, c'_2, c_3, c_4, c_5)$.*

Observons comment le modèle linéaire procèderait pour régler le problème de l'exemple 1. Soit $h = (c_1, c_2, c_3, c_4, c_5)$ et r vide. Nous souhaitons remplacer c_2 par une autre commande c'_2 et revenir ensuite à c_5. Tout d'abord nous devons exécuter une suite d'UNDO jusqu'à obtenir $h = (c_1)$ et $r = (c_2, c_3, c_4, c_5)$. Ensuite, nous pouvons insérer la commande c'_2 mais nous devons après ré-exécuter les commandes c_3, c_4 et c_5 sans utiliser le REDO car la première commande de la *redo list* est celle que nous voulons remplacer.
Une alternative consiste à exécuter les UNDO jusqu'à obtenir $h = (c_1, c_2)$ et $r = (c_3, c_4, c_5)$. À partir de là, il est possible d'exécuter une commande c annulant l'effet de c_2. Nous pouvons donc exécuter c'_2 et ensuite procéder aux REDO nécessaires pour arriver finalement à cet état : $h = (c_1, c_2, c, c'_2, c_3, c_4, c_5)$ et $r = \emptyset$.
Afin de comparer les modèles, nous comptabiliserons pour chacun d'entre eux le nombre d'actions effectuées par l'utilisateur pour résoudre l'exemple. Par action, nous entendons l'exécution ou la modification d'une commande ou d'une méta-commande. Pour ce premier modèle nous arrivons donc à un total de huit commandes (4 UNDO + 4 commandes exécutées). La solution alternative donne également un total de huit.

Cette alternative peut être source de problème. La Figure I.1, extraite du travail de Thomas Berlage [1], représente les états dans lesquels se trouvent l'*history list* et la *redo list* après l'exécution de sept commandes, suivi de l'UNDO des trois dernières. L'état courant se trouve entre c_4 et c_5. À partir de là, si une nouvelle commande est insérée, elle sera ajoutée à l'*history list*, la *redo list* quant à elle reste à disposition et les commandes

qu'elle contient peuvent toujours être ré-exécutées. C'est ce dernier point qui peut poser problème lorsqu'on introduit la notion de dépendance. Reprenons la situation présentée dans la Figure I.1 et plaçons-là dans le contexte d'un programme d'édition d'image : imaginons que c_4 crée un carré et que c_5 modifie la couleur de ce carré. Nous sommes dans un cas où c_5 dépend de c_4. L'état courant étant toujours entre c_4 et c_5, si une nouvelle commande, dont l'effet supprime le carré, y est insérée alors c_5 n'a plus lieu d'être. Or, le modèle autorise quand même la ré-exécution de c_5 alors que l'objet cible n'existe plus. C'est l'un des problèmes récurrents dans la conception d'un modèle d'UNDO, qui, pour le modèle linéaire est résolue au moyen d'une adaptation de celui-ci.

I.2 Restricted Linear Undo model

Afin de pallier à ce problème de dépendance, Berlage [1] introduit une propriété nommée "Stable Execution Property" :

Propriété 1. *Le REDO d'une commande ne peut s'appliquer qu'à partir du même état depuis lequel la commande a été exécutée à l'origine. L'UNDO d'une commande ne peut s'appliquer qu'à partir de l'état qui a été atteint après l'exécution de la commande.*

Autrement dit, si une commande est REDO ou UNDO hors de son contexte d'origine, la propriété est violée. Cette propriété permet d'éviter des cas de REDO de commandes qui n'ont plus lieu d'être. Le Restricted Linear Undo (ou RLU) applique cette propriété au modèle linéaire simple de cette façon : si une nouvelle commande est effectuée alors que la *redo list* est non-vide, le contenu de cette dernière est supprimé.
Représentons le modèle RLU m comme un couple (h, r) avec $h = (c_1, ..., c_m)$ et avec $r = (c_1, ..., c_n)$. L'exécution d'un UNDO et d'un REDO se définit de la même manière que le modèle linéaire. En revanche, $\mathrm{DO}(m, c) = (h', r)$ avec $h' = (c_1, ..., c_m, c)$ et avec $r = \emptyset$.

Voyons comment le modèle RLU pourrait résoudre l'exemple de substitution [Exemple 1]. Comme dans le modèle précédent, nous exécutons quatre UNDO et nous retrouvons l'état où $h = (c_1)$ et $r = (c_2, c_3, c_4, c_5)$. Si nous exécutons c'_2 pour remplacer c_2, la *redo list* est vidée, nous obligeant à ré-exécuter manuellement les dernières commandes pour atteindre l'état où $h = (c_1, c'_2, c_3, c_4, c_5)$, avec $r = \emptyset$. Comme le modèle linéaire, le RLU donne un total de huit commandes (4 UNDO + 4 exécutions).

De nombreuses applications implémentent ce modèle RLU. Nous pouvons par exemple citer Microsoft Word, Microsoft Excel, Notepad++, Photofiltre, etc.

I.3 Script model

Le script model (ou ACS, du nom de ses auteurs [2]), insère les commandes dans un script. Celui-ci consiste en une liste de commandes à exécuter afin d'atteindre un état final depuis un état d'origine. L'utilisateur doit modifier lui-même le script de commandes s'il souhaite obtenir de nouveaux résultats. Le DO ne peut pas être représenté de la même manière que dans les modèles précédents car il ne s'agit pas d'une exécution directe d'une commande car une fois que nous en insérons une dans le script, celui-ci doit être exécuté pour que le commande prenne effet.
Quant à l'UNDO, le script model offre deux stratégies possibles pour son application :

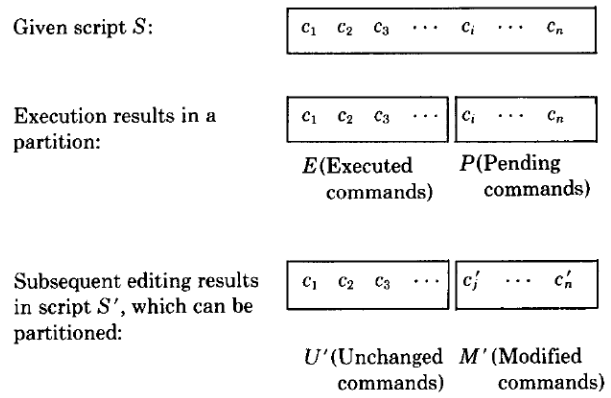


FIGURE I.2 – Script Model [2]

- **Complete Rerun Strategy** : Modifier la ou les commande(s) que l'on souhaite défaire par une nouvelle et ré-exécuter le script depuis l'état d'origine.
- **Checkpoint Strategy** : Défaire les commandes exécutées jusqu'à arriver à un point où la séquence de commandes restantes est un préfixe du nouveau script contenant les modifications et recommencer l'exécution à partir de ce préfixe. La Figure I.2 illustre cette stratégie : le premier niveau représente un script à exécuter. Au second niveau, nous pouvons observer l'exécution du script avec dans le premier rectangle les commandes exécutées et dans le second les commandes en attentes. Après l'exécution complète, une partie du script a été modifiée et nous constatons dans le dernier niveau que le premier rectangle contient les mêmes commandes qu'au niveau précédent, il est donc un préfixe du nouveau script, dont le reste des commandes, situé dans le dernier rectangle, a été modifié.

Afin de savoir qu'elle stratégie est la plus efficace, nous comparons leurs approches. La "Complete Rerun Strategy" possède une approche simple mais inefficace au point de vue temps. En effet plus le script est long, plus le temps d'UNDO et de ré-exécution du nouveau script sera élevé.
La "Checkpoint Strategy" est soit égale, soit plus efficace au point de vue du temps d'UNDO et de ré-exécution. Il sera égal si le script entier est à ré-exécuter mais inférieur dans tous les autres cas. En revanche, cette stratégie consommera de la mémoire car, contrairement à la "Complete Rerun Strategy", des états sont à sauvegarder.

Le REDO ne fait pas partie du script model. La seule façon de REDO une commande anciennement UNDO est de la ré-injecter dans le script avant de le ré-exécuter.

Passons à l'exemple de substitution (Exemple 1) où nous souhaitons substituer la commande c_2 dans $h = (c_1, c_2, c_3, c_4, c_5)$ par une nouvelle commande c'_2. La "Checkpoint Strategy" étant la plus efficace, nous l'appliquons à l'exemple. Dans ce modèle, les commandes ne sont pas dans une liste h mais dans un script S. Pour procéder à la substitution, nous revenons au point où la commande restante c_1 est un préfixe du nouveau script dans lequel nous avons remplacé textuellement c_2 par c'_2. Le nouveau script peut être exécuté et nous obtenons l'état désiré : $S = c_1, c'_2, c_3, c_4, c_5$ avec un total de six actions effectuées par l'utilisateur (4 UNDO + 1 modification + 1 exécution du script). Par cet exemple, nous

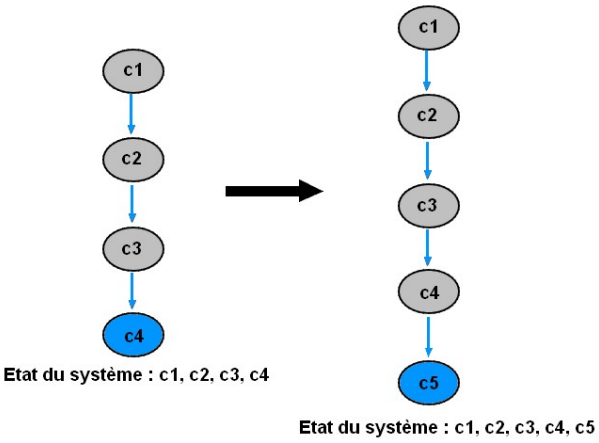


FIGURE I.3 – Exécution d'un DO dans le modèle US&R

montrons que le Script Model ne respecte pas la "Stable Execution Property". En effet, après l'insertion de la commande c'_2, les commandes suivantes dans le script ré-exécuté sont hors de leur contexte original. La propriété est donc violée. Le résultat d'un UNDO dans le script model est l'état qui existerait si la commande défaite n'avait jamais existé.

Le désavantage de ce modèle est que les commandes effectuées par l'utilisateur doivent être éditées textuellement par celui-ci. Les manipulations ne se font pas directement via, par exemple, une interface graphique.

I.4 US&R

Le modèle US&R ou Undo, Skip & Redo, introduit par Vitter, Jeffrey et Scott [7] est représenté par une structure de donnée sous forme de graphe orienté acyclique (DAG). La structure de celui-ci étant complexe dans ce modèle nous le présenterons en simplifiant ses composants. Chacun des nœuds du DAG représente une commande et les arcs indiquent l'ordre dans lequel ils ont été exécutés. Nous avons également une notion de nœud courant qui représente la dernière commande exécutée ainsi qu'une notion de branche active dans laquelle seront ajoutés les nouveaux nœuds.
En plus du DAG, le modèle utilise une notion d'*état du système*. Celui-ci consiste en une séquence de commandes primitives en vigueur et évolue parallèlement avec le DAG. Initialement, l'état du système et le DAG sont vides. Nous représenterons donc le modèle US&R comme $usr = (G, E, current)$ avec G, le graphe, $E = (c_1, ..., c_m, ..., c_n)$ l'état du système et $current$ le nœud courant valant initialement $null$.

L'exécution d'un DO(usr, c) où c est la commande exécutée peut avoir deux effets :

- **Créer un nœud dans la branche active** si $current = c_n$. Dans ce cas, le résultat est un triplet (G', E', c) représentant le modèle modifié avec G' le graphe auquel un nouveau nœud partant de $current$ a été ajouté (la Figure I.3 illustre cet effet du DO, on y observe un premier chemin auquel est ajouté un nouveau nœud, devenant le nœud courant représenté en bleu), $E' = (c_1, ..., c_n, c)$ et où c est également la nouvelle commande courante.

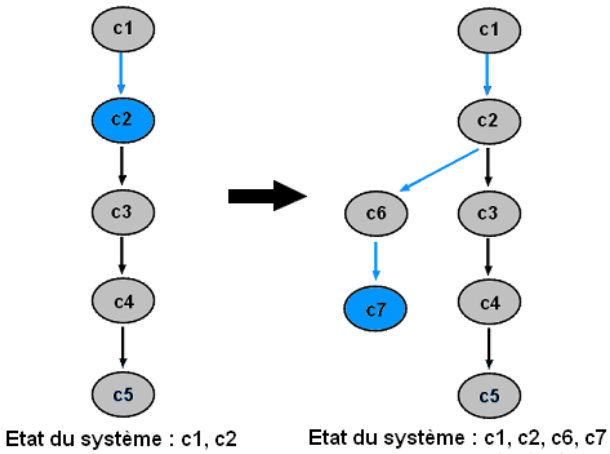


FIGURE I.4 – Exécution d'un DO créant une nouvelle branche dans le modèle US&R

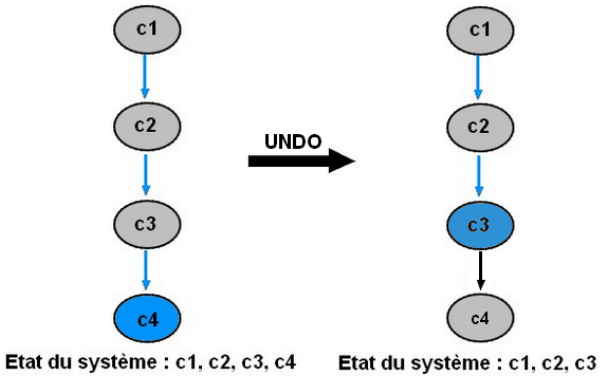


FIGURE I.5 – Exécution d'un UNDO dans le modèle US&R

- **Créer un nœud dans une nouvelle branche** si $current = c_1$ ou $current = c_m$. Dans ce cas, le résultat est un triplet (G', E', c) représentant le modèle modifié avec G' le graphe auquel un nouveau nœud partant de $current$ a été ajouté dans une nouvelle branche (la Figure I.4 illustre cet effet du DO, une nouvelle branche est ajoutée au DAG et est la nouvelle branche active) $E' = (c_1, c)$ si $current = c_1$ ou $E' = (c_1, ..., c_m, c)$ si $current = c_m$ et où c est également la nouvelle commande courante.

La navigation dans le DAG est linéaire et se fait grâce aux différentes méta-commandes UNDO, REDO et SKIP. Cette dernière est propre au modèle US&R et permet de passer une commande c précédemment UNDO. Cette dernière n'est pas supprimée du DAG, mais est marquée "skip". Dans l'état du système, une commande c est marquée "skip" de cette façon : $\boxed{c}$.
L'UNDO sur le modèle usr n'altère pas le graphe mais modifie l'état du système et la commande courante. $\text{UNDO}(usr) = (G, E', c_{n-1})$ avec $E' = (c_1, ..., c_m, ..., c_{n-1})$ l'état du système auquel une commande a été retirée et où c_{n-1} étant la nouvelle commande courante. L'effet de l'UNDO dans le modèle US&R est représenté à la Figure I.5.

L'exécution d'un REDO est plus complexe que l'UNDO pour l'utilisateur car si plusieurs commandes dans le DAG sont disponibles après le nœud courant, il se verra pro-

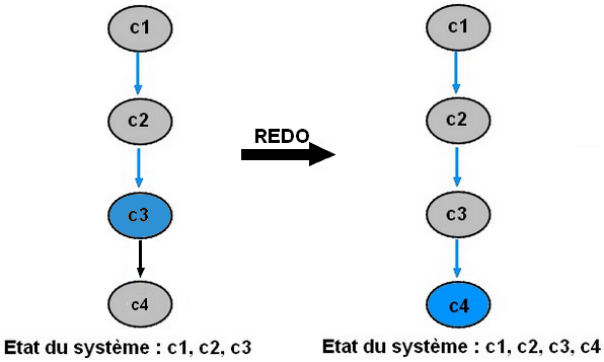


FIGURE I.6 – Exécution d'un REDO dans le modèle US&R

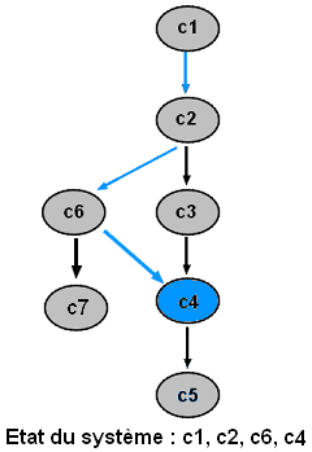


FIGURE I.7 – Exécution d'un REDO de c4 dans le modèle US&R

poser les commandes une à une jusqu'à ce qu'il choisisse laquelle il souhaite REDO. Par exemple, si le nœud courant est parent de deux branches (imaginons $c2$ dans la partie droite de la Figure I.4), nous sommes face à un REDO ambigu et dans ce cas, il sera d'abord proposé à l'utilisateur le nœud de la branche la plus récemment créée et s'il refuse l'autre lui sera proposée. Si toutes les commandes possibles sont épuisées et que l'utilisateur n'en a choisi aucune, le REDO est annulé. Une fois une commande c choisie nous aurons pour le REDO : REDO$(usr, c) = (G, E', c)$ où G est le graphe visible à la Figure I.6, $E' = (c_1, ..., c_m, ..., current, c)$ l'état du système dans lequel c est la nouvelle commande courante. Il se peut que l'utilisateur souhaite REDO une commande d'une branche A à partir d'une branche B, le cas échéant le graphe G s'en retrouverait modifié car un arc serait créé et relierait les deux branches, nous pouvons observer ce cas à la Figure I.7.

La dernière méta-commande du modèle US&R est le SKIP. Tout comme le REDO, l'utilisateur doit choisir la commande souhaitée si plusieurs lui sont proposées. Le SKIP est également sujet à l'ambiguïté si plusieurs commandes sont disponibles lors de son exécution. SKIP$(usr, c) = (G, E', c)$ où G est le graphe inchangé, $E' = (c_1, ..., c_m, ...current, \boxed{c})$ est l'état du système dans lequel la nouvelle commande courante c est marquée "skip", son effet n'est donc pas visible pour l'utilisateur. Dans la Figure I.8 la commande $c3$ a été skippée, cela ne se voit pas dans le graphe mais nous constatons que le skip est bien

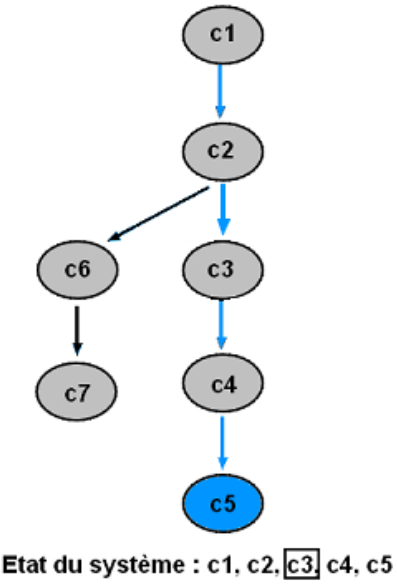


FIGURE I.8 – Graphe après exécution du SKIP de c3 dans le modèle US&R

présent dans l'état du système.

Reprenons l'exemple 1 de substitution. Nous avons l'état du système $E = (c_1, c_2, c_3, c_4, c_5)$. Comme précédemment, nous exécutons les UNDO pour obtenir $E = (c_1)$. Nous devons ensuite exécuter un SKIP de la commande c_2, ce qui produit comme résultat $E = (c_1, \boxed{c_2})$. Nous sommes maintenant libre d'insérer la nouvelle commande c_2' et de REDO les commandes c_3, c_4 et c_5 pour finalement obtenir l'état $E = (c_1, \boxed{c_2}, c_2', c_3, c_4, c_5)$ et l'arbre de la Figure I.9 dans lequel nous observons la nouvelle branche courante contenant la commande c_2 skippée et c_2' active. Le nombre d'actions effectuées est normalement de neuf (4 UNDO + 1 SKIP + 1 exécution + 3 REDO) mais nous allons dans ce modèle-ci prendre en compte le fait que le modèle propose à l'utilisateur quelle commande il souhaite REDO ce qui ajoute une action de validation en plus pour chaque REDO, amenant le total à 12 actions effectuées.

Le modèle US&R ne respecte pas la Stable-Execution Property (Définition 1) car, tout comme le Script Model, le simple fait d'insérer une commande dans l'état du système et d'exécuter ensuite un REDO ne respecte pas la première phrase de la définition. En revanche, d'après les auteurs, le problème posé par la notion de dépendance est pris en compte et certaines contraintes externes font qu'un REDO d'une commande qui n'a pas lieu d'être est interdit, l'utilisateur ne pourra pas REDO la commande si celle-ci lui est proposée.

L'avantage du modèle est sa fonction SKIP, qui offre à l'utilisateur la possibilité de réarranger les commandes, la possibilité d'effectuer un UNDO uniquement sur une commande souhaitée sans en perdre d'autres. Le modèle souffre en revanche, d'une structure complexe et d'un désavantage majeur : les commandes proposées pour les REDO et les SKIP. Cette fonction non seulement n'est pas facile à implémenter mais peut également poser problème à l'utilisateur car plus le nombre de commandes possibles sera élevé, plus le processus peut alors devenir laborieux.

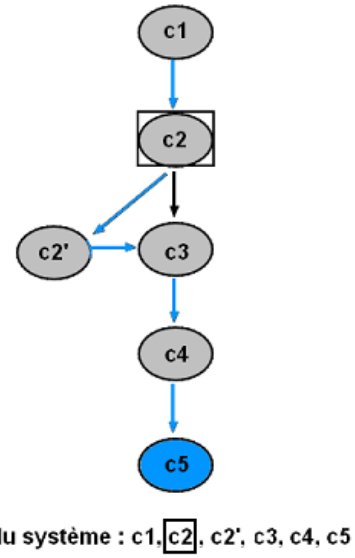


FIGURE I.9 – Exemple de substitution avec le modèle US&R

I.5 Triadic model

À l'instar du modèle US&R, le triadic model [3] possède une troisième méta-commande : le ROTATE. Dans ce modèle, nous avons l'équivalent d'une *history list* et d'une *redo list*, appelées ici respectivement *undo stack* et *redo stack*. Initialement, ces deux *stacks* sont vides. Nous avons donc le modèle $m = (U, R)$ où $U = (c_1, ..., c_n)$ est l'*undo stack* et $R = (c_x, ..., c_z)$ la *redo stack*. Le DO a le même effet que dans le modèle linéaire : lorsqu'une commande c est exécutée, elle est ajoutée à l'*undo stack*. Soit $\text{DO}(m, c) = (U', R)$ avec $U' = (c_1, ..., c_n, c)$. L'UNDO et le REDO se comportent également comme dans le modèle linéaire. Nous avons $\text{UNDO}(m) = (U', R')$ avec $U' = (c_1, ..., c_{n-1})$ et avec $R' = (c_n, c_x, ...c_z)$, ainsi que $\text{REDO}(m) = (U', R')$ avec $U' = (c_1, ..., c_n, c_x)$ et avec $R' = (c_{x+1}, ...c_z)$. La différence avec le modèle linéaire est la présence de la méta-commande ROTATE, qui a pour effet de placer la dernière commande de la *redo stack* en tête de celle-ci. Ceci permet à l'utilisateur de sélectionner quelle commande il souhaite ré-exécuter en appliquant plusieurs fois le ROTATE et en amenant la commande souhaitée en tête de la *stack*. Le ROTATE agit uniquement sur la *redo stack*. Prenons donc R, la *redo stack* comprenant les commandes (c_1, c_2, c_3), après exécution du ROTATE, la *redo stack* est dans l'état (c_3, c_1, c_2) (voir Figure I.10).

Reprenons l'exemple 1 et voyons comment le modèle triadic le gère. Nous avons $U = (c_1, c_2, c_3, c_4, c_5)$ et $R = \emptyset$. Après quatre UNDO et l'insertion de c'_2, nous obtenons $U = (c_1, c'_2)$ et $R = (c_2, c_3, c_4, c_5)$. Étant donné que nous ne souhaitons pas réintégrer c_2, nous devons à présent exécuter trois ROTATE pour que la *redo stack* soit dans l'état (c_3, c_4, c_5, c_2). C'est à ce moment que nous pouvons exécuter trois REDO et arriver à cet état : $U = (c_1, c'_2, c_3, c_4, c_5)$ et $R = (c_2)$. Le nombre d'actions nécessaires pour résoudre l'exemple avec le modèle triadic est de 11 (4 UNDO + 1 exécution + 3 ROTATE + 3 REDO).

L'avantage par rapport au modèle linéaire est que nous pouvons effectuer un UNDO

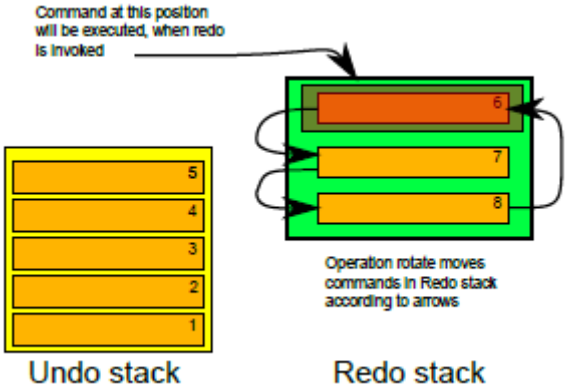


FIGURE I.10 – Représentation du Triadic Model [3]

d'une commande spécifique et puis, grâce au ROTATE, REDO les autres commandes sans REDO celle non-désirée, cette dernière restant dans la *redo stack*. Cet UNDO particulier est appelé l'UNDO sélectif.
En revanche, le ROTATE possède un inconvénient majeur. En effet, à cause de cette méta-commande, le modèle triadic est plus complexe à utiliser car pour effectuer cet UNDO sélectif, l'utilisateur doit, après avoir effectuer les UNDO, abuser de la commande ROTATE afin d'avoir accès aux commandes à REDO. Cela devient laborieux s'il est remonté loin dans l'*undo stack* comme le prouve le total de 11 actions nécessaires pour résoudre l'exemple de substitution (hormis l'US&R, les autres modèles vu jusqu'ici ne dépassaient pas huit actions) avec une *undo stack* de seulement cinq commandes.

I.6 History Tree

Dans l'article de Thomas Berlage [1], une extension du RLU (Section I.2) est présente, dans laquelle l'*history list* et la *redo list* sont remplacé par un arbre nommé *history tree* où chaque nœud est une commande. Nous représentons le modèle m comme $m = (T, current)$ où T est l'*history tree* et où $current$ est la commande courante.
Initialement, $T = \emptyset$ et $current = null$.

Nous définissons le DO comme $\text{DO}(m, c) = (T', c)$ où le couple résultant du DO est le nouveau modèle m' avec T' l'arbre auquel a été ajouté la commande c et où c est également la nouvelle commande courante. Nous observons deux cas possibles quant à T'. Soit, lors de l'exécution aucune commande ne succède à $current$ dans T, dans ce cas c est ajouté en fin de branche (T' est observable à la Figure I.11. Soit, $current$ n'est pas la dernière commande dans la branche et une nouvelle branche contenant c est créée (Figure I.12). Cette solution résout l'inconvénient du RLU qui supprimait la *redo list* lors d'un DO et respecte toujours la SEP.

L'exécution de l'UNDO se fait de la même manière que dans le modèle US&R : nous remontons d'un niveau dans la branche. $\text{UNDO}(m) = (T, current')$ où T n'a pas changé et où $current'$ est la nouvelle commande courante. Une illustration de l'UNDO est observable à la Figure I.13.

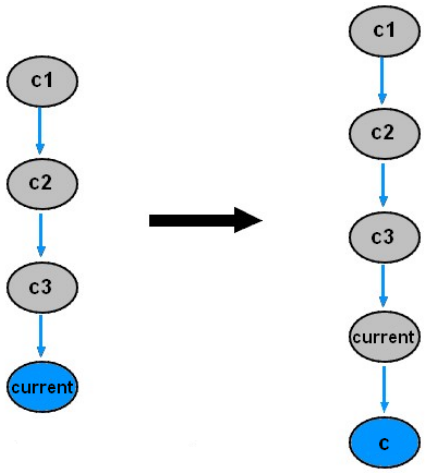


FIGURE I.11 – Évolution de l'arbre T après un DO (1) dans le modèle "history tree"

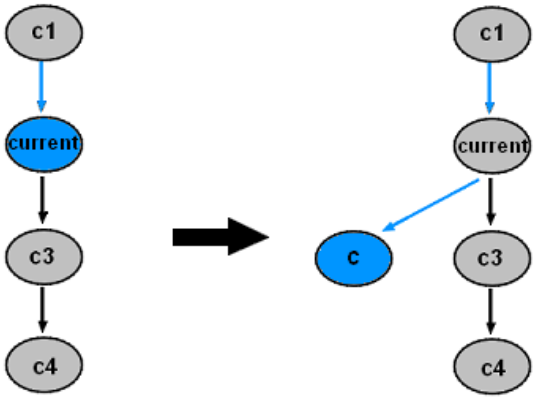


FIGURE I.12 – Évolution de l'arbre T après un DO (2) dans le modèle "history tree"

Grâce à l'*history tree*, aucune commande n'est jamais supprimée et tous les états sont accessibles grâce aux UNDO et aux REDO. Ce dernier est défini comme REDO$(m) = (T, current')$ où T n'a pas changé et où $current'$ est la nouvelle commande courante. Le REDO d'une commande a, comme le DO, deux cas différents, le premier est le cas où le nœud courant ne possède qu'un fils qui après l'exécution, devient le nouveau nœud courant. Le second cas représenté à la Figure I.14 illustre un REDO ambigu. Un REDO ambigu survient lorsque lors d'un REDO, le nœud courant possède plus d'un fils. Si cela survient, l'utilisateur doit décider quelle commande il souhaite ré-exécuter.

Voyons comment l'*history tree* gérerait l'exemple 1. Démarrons avec l'arbre T composé des cinq commandes $c_1, c_2, c_3, c_4 etc_5$ où c_5 est la commande courante. Après quatre UNDO, c_1 devient la commande courante. À ce moment, nous insérons la nouvelle commande c'_2, créant une nouvelle branche composée de c_1 et c'_2 et où c_2 est courant. Nous devons maintenant ré-exécuter manuellement les commandes c_3, c_4 et c_5. Au final, nous obtenons l'arbre de la Figure I.15. Le total d'actions requises par l'utilisateur pour résoudre l'exemple de substitution grâce à l'"history tree" est de huit (4 UNDO + 4 Exécutions).
Nous constatons, en observant la Figure I.15, un inconvénient de ce modèle : l'arbre peut

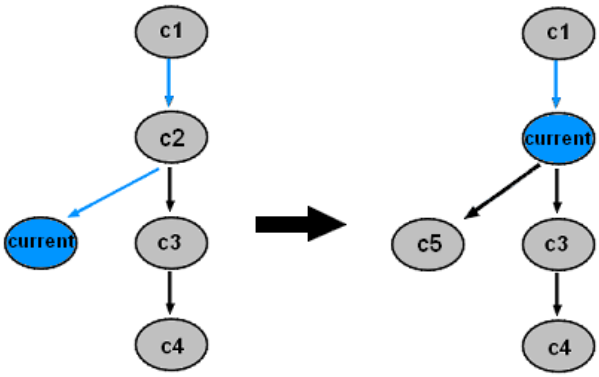


FIGURE I.13 – Évolution de l'arbre T après un UNDO dans le modèle "history tree"

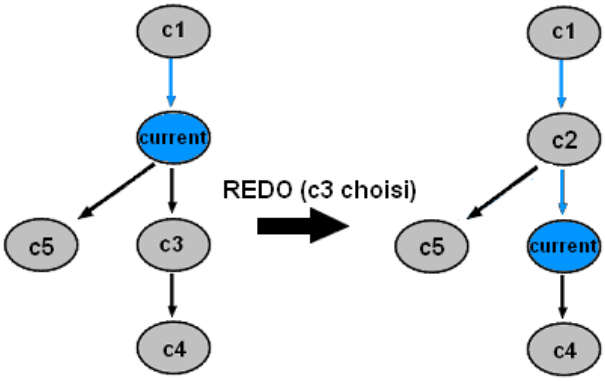


FIGURE I.14 – Évolution de l'arbre T après un REDO ambigu dans le modèle "history tree"

posséder des doublons, cela est directement lié au fait que nous ne pouvons pas REDO des commandes d'autres branches de manière direct. Ce problème est réglé par le modèle de l'UNDO sélectif direct.

I.7 Undo sélectif Direct

L'undo sélectif direct, défini par Thomas Berlage [1] est une amélioration de l'*history tree*. La structure reste la même mais la notion de nœud courant disparait, au profit de celle de chemin courant, c'est-à-dire la séquence de commandes en vigueur. Ce qui différencie principalement ces deux modèles, c'est leur façon de naviguer dans la structure et de sélectionner les commandes pour un UNDO ou un REDO. Dans l'*history tree* la navigation se faisait par méta-commande, de façon linéaire, comme tous les modèles que nous avons présentés jusqu'à maintenant. Ce nouveau modèle permet de sélectionner directement la commande que l'on souhaite UNDO ou REDO quelque soit sa position dans l'arbre. Nous le présenterons comme $m = (T)$ où T est l'arbre de commande, vide à l'origine.

Comme l'*history tree*, deux cas de figures peuvent se présenter lors de l'application de DO(m, c). Soit c s'exécute depuis une commande n'ayant pas de successeur et le résultat est l'arbre T' contenant un nouveau nœud (ce cas de figure est identique à celui de la

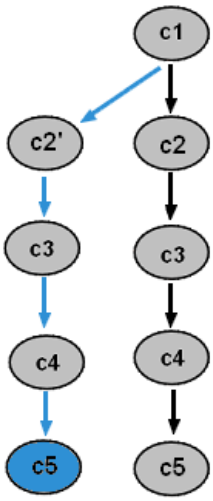


FIGURE I.15 – History Tree après exécution de l'exemple 1

Figure I.11 pour l'*history tree*). Soit c s'exécute depuis un autre nœud et dans ce cas le résultat est l'arbre T' contenant une nouvelle branche, elle-même contenant c (ce cas de figure est une nouvelle fois identique à l'*history tree et peut-être observé à la Figure I.12*). L'exécution d'une méta-commande est, en revanche, différente des modèles précédents. L'UNDO sélectif produit une copie de la commande sélectionnée, exécute son inverse et l'ajoute en fin de liste (Figure I.16). Par inverse, nous entendons une nouvelle commande c' annulant l'effet de la première commande c. Prenons l'UNDO(T, c) où c est la commande sélectionnée, UNDO$(T, c) = (T', c')$ où c' est l'inverse de c et où T' est l'arbre auquel a été ajouté c' (nous observons le résultat à la Figure I.16).

Ce modèle possède également un REDO sélectif : n'importe quelle branche de l'arbre peut-être parcourue afin de REDO une commande, précédemment exécutée (UNDO ou non). La commande sélectionnée est ajoutée à la fin du chemin courant, qui est la branche sur laquelle opère l'utilisateur au moment du REDO. Il est définit comme REDO(T, c) où c est de nouveau la commande sélectionnée, REDO$(T, c) = T'$, l'arbre auquel une copie de c a été ajoutée. Une illustration de cette opération est visible à la Figure I.17.

Afin de montrer la différence entre le modèle sélectif et les autres modèles linéaires présentés, nous utilisons un exemple de T.Berlage [1]. À la Figure I.18 nous observons dans la partie gauche une suite de commandes $(c_1, c_2, c_3, c_4, c_5)$ où c_1 crée un cercle blanc, c_2 colorie ce cercle en gris et c_4 le duplique. Si nous effectuons un UNDO de c_2 avec un des modèles linéaires et que nous ré-exécutons les commandes jusqu'à c_5 nous obtenons le résultat visible dans la partie droite de la Figure I.18, le cercle dupliqué est de la même couleur que l'originale, c'est-à-dire blanc. À présent, observons le résultat obtenu par l'UNDO sélectif de la commande c_2 (Figure I.19), nous constatons qu'une nouvelle commande c_6 a été ajoutée en fin de séquence et que le premier cercle est redevenu blanc. Cela s'explique par le fait que c_6 est une copie de c_2 dont l'effet a été inversé, c_2 transformait le cercle blanc en cercle gris, c_6 transforme donc le cercle gris en cercle blanc.

Par cet exemple, nous démontrons que l'UNDO sélectif direct ne respecte pas la *Stable Execution Property*. Pour contrer les problèmes que cela peut engendrer, tel que le REDO d'une commande n'ayant plus de sens, ce modèle vérifie avant chaque méta-commande si celle-ci peut être exécutée, par exemple en s'assurant que l'objet ciblé existe toujours, si

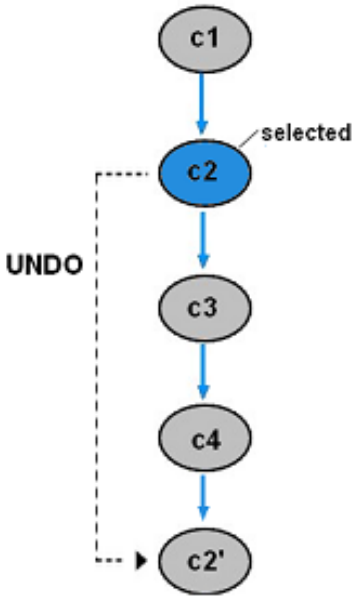


FIGURE I.16 – Effet de l'UNDO sélectif sur l'arbre dans le modèle d'Undo Sélectif Direct

ce n'est pas le cas elle est désactivée. Cette vérification est définie lors de l'implémentation du modèle dans le programme l'utilisant.

Appliquons finalement l'UNDO sélectif direct à l'exemple de substitution 1. Pour rappel, nous souhaitons substituer la commande c_2 par la commande c'_2 dans la liste $(c_1, c_2, c_3, c_4, c_5)$. L'avantage que nous procure l'UNDO sélectif direct par rapport aux autres modèles est que nous pouvons accéder à la commande souhaitée plus rapidement, en une seule action. En effet, dans chacun des autres modèles, il était nécessaire de procéder à quatre UNDO afin que c_2 soit défait. Dans le cas présent, il suffit de sélectionner c_1 et d'insérer c'_2 pour créer une nouvelle branche. Il n'est pas non plus nécessaire de ré-exécuter manuellement les commandes comme pour le RLU ou l'*history tree*, nous pouvons ici simplement sélectionner c_3, c_4 et c_5 et, en assumant qu'elles peuvent être ré-exécutées, les REDO pour arriver au nouveau chemin courant : $(c_1, c'_2, c_3, c_4, c_5)$. La structure correspondante est exposée à la Figure I.20. Le total des actions nécessaires pour résoudre le problème est ici de huit (1 Sélection + 1 Insertion + 3*(1 Sélection + 1 REDO)). Le nombre d'actions est donc équivalent à la plupart des autres modèles mais dans ceux-ci, rares étaient les fois où nous pouvions directement exécuter efficacement un REDO des dernières commandes.

I.8 Undo sélectif en cascade

L'UNDO sélectif en cascade, introduit par Aaron Cass et Chris Fernandez en 2005 [8] est une extension de l'UNDO sélectif direct qui, au lieu d'empêcher l'exécution d'un REDO sur des commandes n'ayant pas lieu d'être, les UNDO jusqu'à atteindre un état d'application stable.

Soit $L = (c_1, c_2, \ldots c_i, c_{i+1}, \ldots c_n)$ une liste de commandes, avec c_{i+1} dépendant de la commande c_i. Lorsque c_i est UNDO, c_{i+1} l'est également.

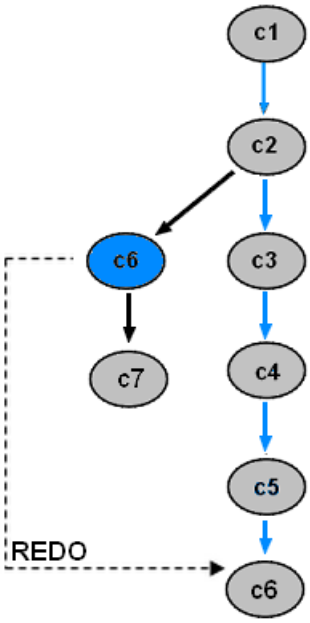


FIGURE I.17 – Effet du REDO sélectif sur l'arbre dans le modèle d'Undo Sélectif Direct

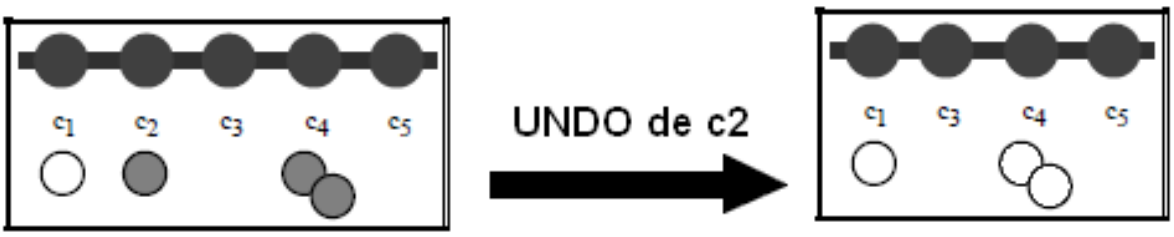


FIGURE I.18 – Exemple des effets des modèles d'UNDO linéaires [1]

Présentons la différence avec le modèle précédent. Pour ce faire, nous utilisons cette suite de commandes [8] :

1. Insérer "Hello"
2. Mettre "Hello" en italique
3. Copier "*Hello*"
4. Le coller à un endroit x

Les deux "Hello" sont donc en italique. Si nous exécutons un UNDO sur la seconde commande avec l'UNDO sélectif direct [1], le "Hello" original est de nouveau en forme normale et la copie est toujours en italique. Tandis que si nous exécutons la même opération mais avec comme modèle l'UNDO sélectif en cascade, les deux "Hello" reviennent à une typographie normale. Le résultat implique que la mise en italique ne s'est jamais produite.

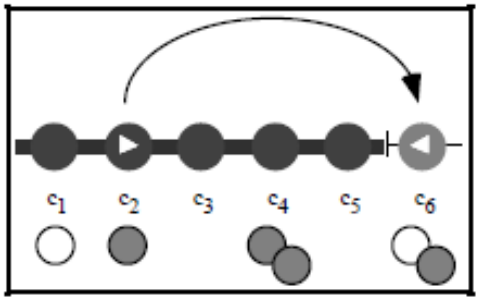


FIGURE I.19 – Exemple de l'effet de l'UNDO sélectif [1]

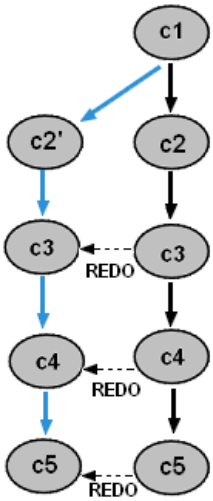


FIGURE I.20 – Structure du modèle de l'UNDO sélectif direct pour l'exemple

I.9 Étude comparative

Maintenant que nous connaissons les modèles d'UNDO existants, nous nous interrogeons sur leur utilisabilité : sont-ils vraiment intuitifs pour les utilisateurs ? Les résultats qu'ils produisent correspondent-ils à leurs attentes ? Afin d'obtenir des réponses, Cass et Fernandez ont présenté une étude comparative impliquant le modèle d'UNDO sélectif en cascade, le modèle linéaire (Section I.1)et le *script model* (Section I.2) [4]. Ils ont présenté l'étude à un panel de 28 personnes. Celle-ci consistait dans un premier temps à réaliser 4 étapes simples (Figure I.21) :

1. Dessiner un cercle
2. Dessiner un carré
3. Dessiner un triangle
4. Colorier le cercle

Ensuite, une deuxième éatpe consistait à demander aux participants de dessiner ce que l'image deviendrait si on exécutait un UNDO sur l'étape 4. Pour le panel entier, le résultat était clair : l'intérieur du cercle n'est plus colorié (Fig I.22).

Lors de la dernière étape, ils demandaient aux participants de dessiner ce qui se passerait si un UNDO était exécuté sur l'étape 1 (Dessiner le cercle).

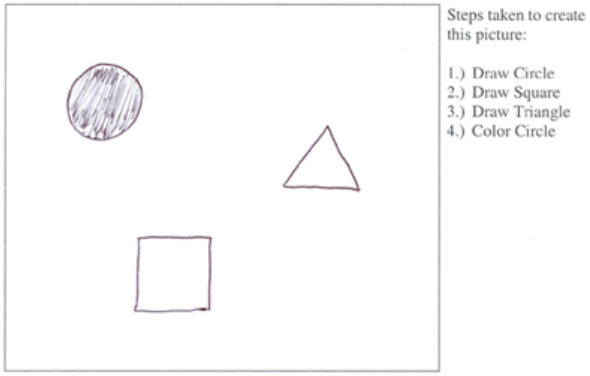


FIGURE I.21 – Exercice de l'étude de Cass et Fernandez [4]

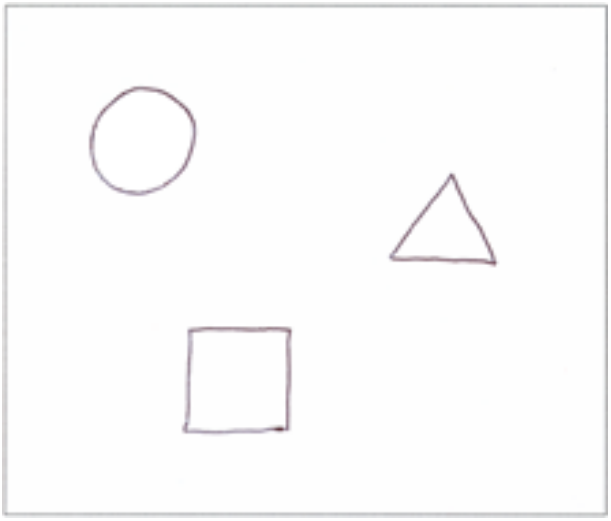


FIGURE I.22 – Résultat du premier exercice [4]

Trois résultats différents se sont dégagés (Figure I.23), chacun correspondant à l'application d'un modèle d'UNDO particulier :

1. Linéaire : On part de la dernière étape et on les défait toutes jusqu'à l'étape 1 comprise.
2. Sélectif en cascade : On défait le cercle, il n'y a plus de raison de le colorier.
3. Script : On défait tout jusqu'à l'étape 1 comprise, et on ré-exécute le reste.

Observons quels auraient été les résultats avec les modèles qui ne correspondent à aucun ci-dessus. Précisons que, bien que le *script model* procède à une ré-exécution des commandes, nous ne considérons que la partie UNDO du modèle et non le REDO, la ré-exécution faisant partie de la stratégie d'UNDO du *script model*.

RLU : Nous pouvons affirmer que le résultat est identique au modèle linéaire.

US&R/Triadic/History Tree : Avant de défaire le cercle nous devons défaire toutes les autres étapes. Le résultat est linéaire.

Undo Sélectif Direct : Sélection puis UNDO de la commande 1. Celle-ci étant "Dessiner un cercle" , l'inverse devrait être "Effacer le cercle". Comme le résultat de la commande était un contour de cercle, l'inverse efface celui-ci et il nous reste

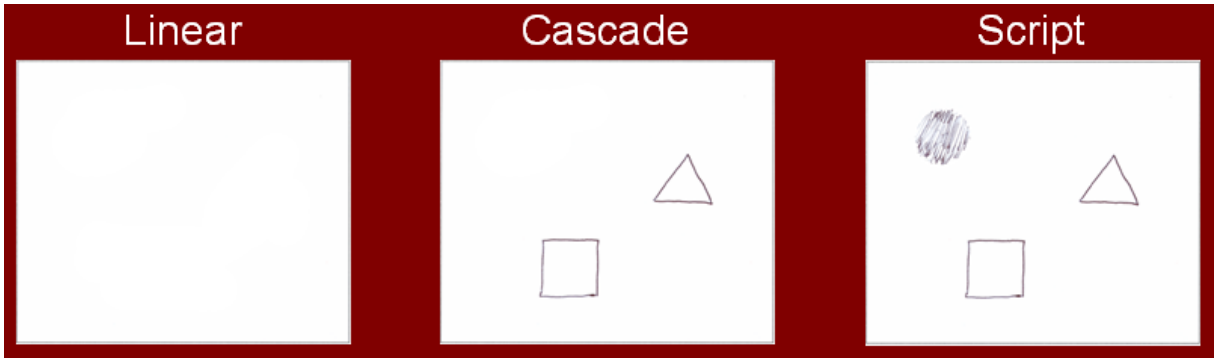


FIGURE I.23 – Résultat de l'exercice de l'étude [4]

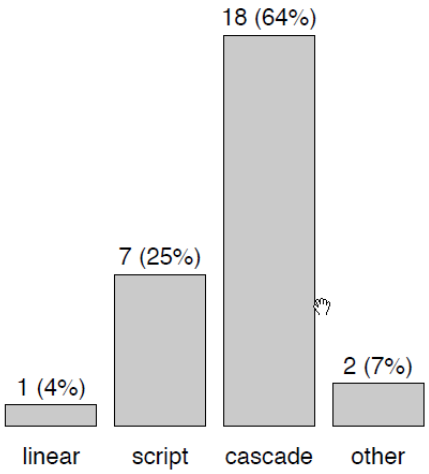


FIGURE I.24 – Résultat de l'étude [4]

la partie intérieure coloriée du cercle, autrement dit le résultat correspondant au *script model*.

Les résultats de l'étude sont représenté sur la Figure I.24. Bien que la majorité des modèles produisent un résultat linéaire sur cet exercice, l'étude révèle que la majorité des utilisateurs s'attend à un résultat correspondant au modèle en cascade lorsqu'ils exécutent un UNDO.

I.10 Synthèse

Nous avons présenté huit différents modèles d'UNDO existants. Revoyons quels en sont leurs avantages et inconvénients (Tableau I.1) :

Table I.1 – Tableau récapitulatif des modèles

Modèle	Avantages	Inconvénients
Linéaire	Simple à utiliser. Facile à mettre en œuvre.	Ne respecte pas la Stable Execution Property (SEP). La liste peut devenir très longue car aucune commande n'est supprimée.
Restricted Linear	Simple à utiliser. Facile à mettre en œuvre Respecte la SEP.	REDO limité à cause de la SEP. *Redo list* supprimée lors d'une insertion de commande.
Script	Permet l'UNDO sélectif	Ré-exécution à chaque modification Les commandes doivent se faire textuellement, dans un script.
US&R	Fonction SKIP. Permet l'UNDO sélectif	L'UNDO sélectif ne se fait pas de manière directe. Structure complexe. REDO et SKIP peuvent être laborieux à mettre en œuvre et à utiliser. Ne respecte pas la SEP.
Triadic	Fonction ROTATE puissante Facile à mettre en place Permet l'UNDO sélectif	L'UNDO sélectif ne se fait pas de manière directe. ROTATE peut être laborieux à utiliser Ne respecte pas la SEP.
History Tree	Résout l'inconvénient du RLU. Aucune commande n'est supprimée.	L'arbre peut devenir vaste. Possibilité de doublons.
Sélectif	Accès direct aux commandes Sauvegarde de toutes les commandes Certaines commandes ne seront pas accessibles aux UNDO/REDO si elles ne sont pas censé l'être. REDO de n'importe quelle commande possible.	Toutes les commandes, même celles qui n'ont plus de sens font toujours partie de l'arbre. Ne respecte pas la SEP.
Sélectif en cascade.	Accès direct aux commandes. Intuitif pour la majorité des utilisateurs. Supprime les commandes n'ayant plus de sens.	Ne respecte pas la SEP.

Chapitre II

Évaluation des modèles d'UNDO pour la configuration

Nous venons de présenter les différents modèles d'UNDO existants. Cela dans le but de trouver une solution au problème d'UNDO dans la configuration consistant à trouver une solution permettant à l'utilisateur de naviguer dans les configurations réalisées et d'UNDO/REDO certaines commandes s'il le souhaite. Afin d'identifier les modèles qui pourraient être exploitables pour résoudre le problème, nous allons les comparer sur base de critères que le modèle final devra respecter.

II.1 Critères

1. Tout d'abord le modèle doit être efficace et doit offrir la possibilité de comparer plusieurs alternatives. Nous souhaitons que les utilisateurs se servant du configurateur puissent tester plusieurs configurations sans devoir ré-exécuter les séquences de commandes après chaque comparaison. Une fois qu'un utilisateur est retourné à un état antérieur a depuis un état o, tous les états existants lorsque l'utilisateur était dans o doivent toujours exister ou être accessibles lorsque a est atteint. Par "être accessibles", nous entendons que l'utilisateur doit pouvoir les atteindre sans devoir manuellement ré-exécuter les commandes nécessaires. Nous avons donc un premier critère d'**efficacité**.
2. Ensuite, l'utilisateur doit pouvoir naviguer dans ses configurations avec efficience. Une fois que des commandes ont été effectuées, il devrait pouvoir naviguer à sa guise dans les différentes configurations avec le moins d'efforts possibles. Nous entendons par là que la navigation ne soit pas linéaire. Nous dégageons ici un critère d'**efficience dans la navigation**. Nous mesurerons ce critère grâce aux nombres de commandes effectuées par chaque modèle pour résoudre le problème de substitution (Exemple 1) introduit dans le chapitre I et nous tirerons des conclusions de leurs efficiences en comparant ces résultats en fin de chapitre.
3. Enfin, nous recherchons un modèle qui puisse **s'appliquer à la configuration**. Il faut qu'il soit capable de tenir compte des contraintes que cela apporte comme par exemple le fait qu'en fonction de certains choix de l'utilisateur, d'autres options ne soient plus valides par après ou, au contraire, qu'elles soient les seules valides. Il est également nécessaire que l'application d'un UNDO ou d'un REDO respecte cette

gestion de contraintes, c'est à dire qu'après un UNDO, les contraintes provoquées par la commande d'origine sont également UNDO. Il en va de même pour le REDO. Une cohérence doit également être respectée, si nous effectuons un UNDO sur deux choix successifs mais que le second dépend de l'existence du premier, alors il doit être impossible d'exécuter un REDO sur le second tant que le premier ne l'a pas lui-même été. Une propriété définie par Karel Jakubec [3] comme une "*Weakened Stable Execution Property*" ou SEP affaiblie s'applique particulièrement à ce critère. Cette propriété a été définie car Jakubec jugeait la SEP (Propriété 1) trop stricte et n'étant vraie que pour un nombre trop limité de modèles.

Propriété 2. *Pour REDO une commande C, toutes les commandes dont C dépend sont REDO avant C. Pour UNDO une commande C, toutes les commandes dépendant de C sont UNDO avant C.*

Nous considérerons donc le critère d'applicabilité à la configuration respecté si le modèle respecte la "*Weakened SEP*".

Analysons à présent les modèles existants et voyons s'ils respectent ces critères.

II.2 Modèle Linéaire

1. **Efficacité :** × Il n'est pas possible avec le modèle linéaire de tester plusieurs configurations sans devoir défaire les états déjà existants et sans devoir ré-exécuter manuellement les commandes.
2. **Efficience :** × Le modèle linéaire résout le problème de substitution en huit commandes (4 UNDO + 4 Commandes) (Section I.1) mais le système de navigation est linéaire et ne respecte donc pas le critère.
3. **Applicabilité à la configuration :** ✓ Le modèle linéaire respecte la "*Weakened SEP*". Dans ce modèle, tout ce qui est effectué, UNDO ou REDO se fait dans un ordre précis. Si a est effectué avant b, et que b dépend de a, les UNDO déferont d'abord b puis ensuite a et il n'y aura pas moyen que b soit REDO avant a.

II.3 Restricted Linear Undo model

1. **Efficacité :** × Tout comme le modèle linéaire, le RLU ne permet pas de comparer plusieurs configurations différentes car il n'est pas possible d'enregistrer plusieurs séquences de commandes différentes sans d'abord UNDO ce qui existe. En plus de cela, la suppression du contenu de la *redo list* empêche de comparer simultanément plusieurs configurations.
2. **Efficience :** × Le RLU résout aussi le problème de substitution en huit commandes (4 UNDO + 4 Commandes) (voir Section I.2) mais le système de navigation est linéaire et ne respecte donc pas le critère.
3. **Applicabilité à la configuration :** ✓ Le système du RLU étant le même que la modèle linéaire, il respecte également le critère. Le fait que le RLU supprime la REDO list lors de l'insertion d'une nouvelle commande n'affecte en rien le respect de la "Weakened SEP".

II.4 Script Model

1. **Efficacité :** × La comparaison de plusieurs configurations simultanément est impossible avec le script model. En effet, pour construire plusieurs configurations, plusieurs séquences de commandes sont nécessaires. Si chacune de ces séquences est un script, il faudra les UNDO et les relancer à chaque fois, défaisant ce qui existe déjà. Le critère n'est pas respecté.
2. **Efficience :** × Le script model résout l'exemple de substitution en six actions (voir Section I.3) mais le modèle ne possède pas de système de navigation à proprement dit étant donné que toute commande est insérée manuellement dans un script puis que celui-ci est entièrement exécuté et doit l'être de nouveau à chaque modification souhaitée. Nous concluons donc que cela ne satisfait pas le critère d'efficience.
3. **Applicabilité à la configuration :** × Le script model laisse une grande liberté à l'utilisateur quand à l'ordre des exécutions des commandes. Une commande b dépendant d'une commande a peut donc être placée dans le script pour exécution avant a. La "*Weakened SEP*" n'est pas respectée par le script model.

II.5 US&R

1. **Efficacité :** ✓ Le modèle US&R enregistre tous les états et autorise plusieurs séquences de commandes (branches), ce qui permet de se rendre à d'autres configurations tout en gardant les autres états accessibles.
2. **Efficience :** × Le modèle US&R résout l'exemple de substitution en 12 actions d'utilisateur, ce qui en fait le modèle le moins efficient jusqu'à présent (Section I.4). Ajoutons le fait que la navigation est linéaire et le critère n'est pas respecté.
3. **Applicabilité à la configuration :** ✓ Soit des commandes a, b et c exécutées dans cet ordre et c dépend de b. Si nous effectuons deux UNDO suivit d'un SKIP de la commande b, nous avons l'opportunité de REDO la commande c, ce qui violerait la "Weakened SEP". Or, d'après les auteurs [7], si une commande dépend d'une ancienne et que cette dernière n'est plus en vigueur, alors la première commande est déclarée "non-exécutable" si l'utilisateur souhaite la REDO. Cela préserve la "Weakened SEP".

II.6 Triadic Model

1. **Efficacité :** ✓ Le triadic model permet de comparer plusieurs configurations car toutes les commandes sont sauvées et peuvent être récupérées dans la *redo stack* grâce à la commande ROTATE. Chaque commande est donc toujours accessible. Nous affirmons d'ailleurs dans la Section I.5 que ce modèle permet l'UNDO sélectif.
2. **Efficience :** × Le modèle triadic résout l'exemple de substitution en 11 actions d'utilisateur. Ce résultat supérieur à la plupart des autres modèles est dû à la présence de la méta-commande ROTATE (voir Section I.5). Une nouvelle fois la navigation est linéaire, compromettant le critère d'efficience.
3. **Applicabilité à la configuration :** × La fonction ROTATE du triadic model enfreint la "*Weakened SEP*". En effet, le ROTATE permet de sélectionner une

commande de la *redo list* et de la REDO même si cette commande dépend de l'existence d'une autre commande de la *redo list*.

Nous remarquons que les deux modèles possédant une méta-commande supplémentaire (SKIP pour l'US&R et ROTATE pour le triadic model) ont une efficience réduite par rapport aux autre modèles. Ils résolvent l'exemple de susbstitution en un nombre d'actions supérieur à ceux-ci.

II.7 History Tree

1. **Efficacité :** ✓ Grâce à l'History Tree, nous pouvons comparer plusieurs alternatives de configurations sans perdre ce qui a été réalisé précédemment. En effet, toutes les commandes sont sauvegardées et elles sont toujours accessibles peut importe l'endroit où nous sommes situés dans l'arbre.
2. **Efficience :** × Huit actions sont une nouvelle fois nécessaire au modèle pour résoudre l'exemple de substitution (voir Section I.6). Pour naviguer dans l'history tree, l'utilisateur doit utiliser des UNDO et des REDO et passer d'un nœud à l'autre : c'est un système linéaire. L'efficience n'est pas atteinte.
3. **Applicabilité à la configuration :** ✓ Le système de navigation de l'History Tree est linéaire. Si les commandes a, b et c sont exécutées dans cet ordre, elles seront UNDO dans l'ordre c, b, a et REDO dans le même ordre dans lequel elles ont originellement été exécutées. La dépendance entre les commandes ne posent ici aucun problème, la propriété est respectée.

II.8 UNDO sélectif direct

1. **Efficacité :** ✓ L'UNDO sélectif permet l'accès direct à n'importe quelle commande dans le graphe et permettrait donc de comparer différentes configurations en un clic. Cela sans jamais perdre les autres configurations créées entretemps.
2. **Efficience :** ✓ Huit actions sont nécessaire au modèle pour résoudre l'exemple de substitution (Section I.7). La navigation ici se fait de manière direct, en un clic. Nous avons un système de navigation efficient, il est en effet plus aisé pour un utilisateur de choisir directement quelle commande atteindre plutôt que de s'y rendre en usant de méta-commandes.
3. **Applicabilité à la configuration :** × Comme nous l'avons mentionné dans l'état de l'art à la section I.7, l'UNDO sélectif direct ne respecte pas la SEP. Une vérification est effectuée lorsque l'utilisateur exécute un REDO, afin de s'assurer que l'on ne REDO pas une commande n'ayant plus lieu d'être. En revanche, le modèle autorise l'UNDO de n'importe quelle commande dans l'arbre puisque l'effet de l'UNDO est d'effectuer l'inverse de la commande. Par exemple si une commande a crée un cercle et qu'une commande b colorie ce cercle, si nous effectuons un UNDO sur a, une commande a' sera créée, supprimant ce cercle mais b existera toujours dans l'arbre, elle ne subira pas d'UNDO comme cela aurait dû être le cas pour respecter la "Weakened SEP".

II.9 UNDO sélectif en cascade

1. **Efficacité :** ✓ Le système de navigation est le même que l'UNDO sélectif. Nous pouvons donc accéder à n'importe quelle configuration dans le graphe afin de les comparer, encore une fois sans perdre les autres configurations.
2. **Efficience :** ✓ Comme l'UNDO sélectif, huit actions sont nécessaire au modèle pour résoudre l'exemple de substitution (voir Section I.8). La navigation se fait de la même manière que dans le modèle précédent ; le système de navigation est efficient.
3. **Applicabilité à la configuration :** ✓ L'UNDO sélectif en cascade a le même système que l'UNDO sélectif direct avec l'atout supplémentaire qu'il UNDO également les commandes dépendantes de celle visée. Cet avantage comble le problème empêchant l'UNDO sélectif direct de respecter la "Weakened SEP".

Nous pouvons ajouter, pour ce modèle, le fait que le résultat d'un UNDO sélectif en cascade est ce à quoi s'attend la majorité des utilisateurs d'après l'étude de Cass et Fernandez présenté en Section I.9. À la différence de l'UNDO sélectif direct, toutes les commandes ne sont pas sauvegardées, l'effet cascade supprimant les commandes dépendant de commandes antérieurs ayant été UNDO.

II.10 Conclusion de l'analyse

Le tableau récapitulatif II.1 résume le respect des critères pour chaque modèle. Pour rappel : l'efficacité est respectée si lors de la navigation, tous les états restent accessibles (Section II.1). L'efficience est respectée si la navigation du modèle n'est pas linéaire. Dans le tableau, nous indiquons également entre parenthèses le nombre de commandes nécessaires au modèle pour résoudre l'exemple de substitution. Enfin le troisième critère est respecté si le modèle satisfait la propriété de la "Weakened SEP" (Propriété 2).
Dans le tableau récapitulatif nous observons que tous les modèles possédant une structure en arbre respectent le critère d'efficacité et que seul les modèles permettant une sélection direct respectent le critère d'efficience. Nous concluons que ce sont deux aspects que nous allons devoir réutiliser lors du développement de notre propre modèle pour la configuration. Nous observons également qu'il n'y a que l'UNDO sélectif en cascade qui respecte nos trois critères. Nous nous baserons donc sur ce modèle pour proposer notre propre solution.

TABLE II.1 – Récapitulatif de l'analyse

	Efficacité	Efficience	"*Weakened SEP*"
Linéaire	×	× (8)	✓
RLU	×	× (8)	✓
Script	×	× (6)	×
US&R	✓	× (12)	✓
Triadic	✓	× (11)	×
History tree	✓	× (8)	✓
Undo Sélectif Direct	✓	✓ (8)	×
Undo Sélectif Direct en Cascade	✓	✓ (8)	✓

Chapitre III

Le Modèle D.A.C

Sur base des résultats de l'évaluation des modèles, nous pouvons à présent proposer notre propre modèle d'UNDO pour la configuration. Dans un premier temps nous définirons d'abord les notions de configurations dont nous tiendrons compte dans notre définition du modèle. Nous baptisons celui-ci le modèle D.A.C pour DAG, Assignment & Command

III.1 La configuration

Nous allons définir les notions de configurations afin de savoir quels aspects nous allons devoir prendre en compte dans le développement de notre modèle. Tout d'abord, les différentes variables présentées à l'utilisateur ne sont pas toutes de mêmes types et les valeurs qu'elles peuvent prendre sont définie par une fonction de typage.

Définition 1. *Soit V un ensemble de variables. Une* ***fonction de typage*** *sur V est une fonction totale $\tau : \tau(v \in V) = dom(v)$ associant à toute variable le domaine des valeurs auxquelles elle peut être assignée.*

Nou considérons que le domaine des valeurs possibles pour une variable est parmi les suivants :

- Booléen (true, false ou nil)
- $\mathbb{Z}^+ \cup \{Nil\}$
- Un ensemble de valeurs dites "énumérées", incluant nil [1].

Connaissant son domaine, une variable peut être assignée à une valeur de celui-ci.

Définition 2. *Soit v une variable et dom(v) son domaine. Une* ***assignation*** *de v est un couple (v, val) avec val ∈ dom(v).*

Toute assignation n'est généralement pas autorisée dans un configurateur, que ce soit pour des raisons techniques, physiques, commerciales, ou autres. Ainsi, des contraintes sont définies entre les variables pour empêcher certaines (combinaisons d')assignations.

1. Nil = Pas de valeur

Définition 3. *Soit $V = \{v_1, v_2, \ldots, v_n\}$ un ensemble fini de variables et τ une fonction de typage. Une* ***contrainte*** *sur V et τ est une fonction totale $c : dom(v_1) \times dom(v_2) \times \ldots \times dom(v_n) \rightarrow \{\top, \bot\}$.*

Une contrainte définit ainsi si un ensemble (complet ou non) d'assignations est acceptable ou non.
Grâce à ces premières définitions, nous pouvons présenter ce qu'est un modèle de configuration.

Définition 4. *Un* ***modèle de configuration*** *est un ensemble de variables V, une fonction de typage $\tau : V \rightarrow dom(V)$ (où Types est un ensemble de types) et un ensemble de contraintes C sur V et τ.*

Lors d'une assignation, à cause des contraintes du modèle, une variable pourrait n'avoir qu'une seule valeur autorisée. Pour faciliter le travail de l'utilisateur, un configurateur assigne cette valeur à la variable concernée de manière automatique. Ce mécanisme se nomme propagation. Chaque nouvelle assignation peut impliquer une ou plusieurs propagations. Ainsi, on nomme configuration un ensemble d'assignations et de propagations résultantes.

Définition 5. *Soit V un ensemble de variables et τ une fonction de typage. Une* ***configuration*** *est un ensemble $\{(v_i, val_i) \mid v_i \in V \wedge val_i \in dom(v_i)\}$ d'assignations et de propagations tel que $v_i = v_j \rightarrow i \neq j$. Soit c, une configuration et m, un modèle de configuration, c peut satisfaire ou non les contraintes de m.*

En plus des critères définit en section II.1 nous ajouterons le fait qu'un nouveau modèle doit prendre en compte le fait que le configurateur utilise des assignations et non pas des commandes. Une commande reprend n'importe quelle action de l'utilisateur tandis qu'une assignation est une commande associant une valeur à une variable. Nous constatons également que le configurateur gère lui-même les contraintes grâce au moteur de configuration ; le critère d'applicabilité à la configuration est donc à modifier étant donné que le modèle d'UNDO n'a plus à en tenir compte. En revanche, il doit tenir compte des propagations pouvant être engendrées par les assignations.
Enfin, grâce au nouveau modèle, l'utilisateur devra pouvoir :

- sélectionner n'importe quelle configuration existante à n'importe quel moment et depuis n'importe quelle autre alternative.
- UNDO et REDO les assignations.
- explorer et comparer plusieurs alternatives
- construire de nouvelles alternatives de configurations à partir d'existantes.

III.2 Modèle D.A.C

Afin de respecter les critères et d'atteindre les objectifs que nous venons de fixer, nous reprendrons les concepts de précédents modèles tels que l'arbre de l'UNDO sélectif afin d'offrir une vue d'ensemble des configurations produites à l'utilisateur, ainsi que les UNDO et REDO list qui contiendront les historiques des commandes exécutées. Le configurateur utilisant des assignations, nous intégrerons également celles-ci dans le nouveau modèle. Nous appelons celui-ci le modèle D.A.C pour DAG, Assignment, Command.

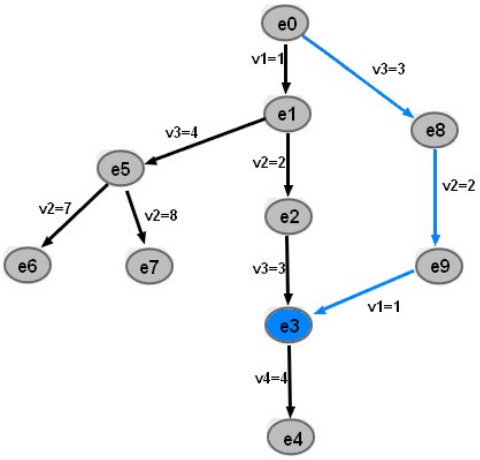


FIGURE III.1 – Exemple du modèle D.A.C

Nous avons choisi d'utiliser un DAG (Directed Acyclic Graph) pour la représentation des assignations à l'utilisateur. Un DAG possède une plusieurs nœuds dont un nœud racine.

Définition 6. *Un nœud $n(c)$ symbolise une configuration c. Initialement nous avons un nœud racine r dont la configuration c est une configuration vide.*

Les nœuds dans le DAG sont reliés par des arcs.

Définition 7. *Un arc $arc(a, o, t)$ représente une assignation a avec o le nœud d'origine et t le nœud cible relié par cet arc.*

Une suite d'arcs dans le graphe forme un chemin.

Définition 8. *Soit un chemin $path(arcs)$ avec $arcs$ une liste d'arcs ordonnée selon la contrainte suivante : pour chaque arc a dans $arcs \rightarrow a.t = a_{+1}.o$. Pour chaque chemin, le premier arc a_0 doit avoir comme nœud d'origine le nœud racine r.*

Nous obtenons finalement la définition d'un DAG :

Définition 9. *Soit $DAG(Paths, r, c, cp)$ avec $Paths$ la liste de chemins, r le nœud racine, c le nœud courant qui est le nœud à partir duquel sera exécuté la prochaine assignation et cp le chemin courant qui est le chemin sur lequel l'utilisateur travaille.*

Un exemple de DAG est montré en Figure III.1. Dans celui-ci nous observons que plusieurs configurations différentes ont déjà été réalisées, résultant en plusieurs chemins différents. Le nœud $e0$ est le nœud racine, le nœud courant est identifiable à sa couleur bleue, ici le nœud $e3$. Le chemin courant est également identifiable grâce à sa séquence d'arcs bleus.

Dans le modèle D.A.C, une autre structure se joint au DAG : les UNDO et REDO lists. Le concept de DAG permettant de sélectionner directement la configuration souhaitée et d'avoir une vue d'ensemble du travail accompli, est ici combiné avec un UNDO linéaire et un REDO sélectif. Les *undo* et *redo lists* associées au DAG sont un soutien pour l'utilisateur : l'*undo list* lui permet d'avoir un historique de ses commandes et lui permet

de les UNDO s'il le souhaite et la *redo list* lui permet de savoir quelles sont les commandes UNDO qui peuvent directement être ré-exécutées sans avoir à le faire manuellement. Nous représentons donc notre modèle comme suit :

Définition 10. *Soit dac, le modèle définit comme* $dac = (D, U, R)$ *où* D *est le DAG,* U *est l'undo list* $(c_1, ..., c_m)$ *et* R *la redo list* $(c_n, ..., c_p)$.

Nous pouvons appliquer quatre commandes sur le modèle D.A.C : le saut, l'assignation, la désassignation et la réassignation. Nous définissons l'application de DO sur *dac* comme un couple (dac, c) où c est la commande exécutée et où après exécution, nous obtenons dac' où $dac' = (D', U', R)$ avec U' l'*undo list* à laquelle a été ajoutée c : $(c_1, ..., c_m, c)$. D' est le DAG ayant subi l'effet de c. Celui-ci est différent en fonction de la commande effectuée.

Le saut est une commande consistant à changer de nœud courant. Elle est représentée dans les historiques comme $j(o, t)$ où o *(origin)* est l'ancien nœud courant et t *(target)* est le nouveau nœud courant. Afin d'éviter de s'encombrer d'une *redo list* trop importante, le saut a pour effet de supprimer celle-ci. Bien que le REDO soit sélectif et donc que l'utilisateur puisse sélectionner n'importe quelle commande de la liste, le configurateur vérifie toujours si une assignation peut se faire et si la configuration résultante est valide, ce qui empêche d'obtenir des états n'ayant pas de sens comme nous l'avons vu en Section I.7.

En tant que commande, une assignation est représentée dans l'*undo list* et la *redo list* comme tel : $a(v, val, o, t)$ où v est la variable assignée et val sa valeur, et où o est le nœud d'origine et t le nœud cible, afin que l'utilisateur puisse savoir de quel nœud il s'agit lorsqu'il souhaite undo ou redo une commande. Dans le DAG, les assignations sont représentées par les arcs et sont indiquées comme "$v = val$", plus intuitif pour l'utilisateur. Certaines contraintes sont posées sur ces assignations afin de prévenir une structure complexe du DAG :

- Une assignation est unique dans un chemin donné.
- Une variable ne peut être assignée qu'une seule fois dans un chemin donné.

Le résultat d'une assignation $v = val$ est une configuration c dans laquelle la variable v est assignée à la valeur val. Plusieurs situations sont possibles lors d'une assignation, en fonction de la situation l'effet qu'a l'assignation sur le DAG est différent. Une première catégorie de situations possibles est celle dépendant de la configuration c résultant de l'assignation :

- c **n'existe pas et le nœud courant ne possède pas de successeurs (Figure III.2)**. Dans ce cas un nouveau nœud représentant c est créé ainsi qu'un nouvel arc dont le nœud d'origine est le nœud courant avant assignation et dont le nœud cible est le nouveau nœud, également nouveau nœud courant.
- c **n'existe pas et le nœud courant a au moins un successeur (Figure III.3)**. Dans ce cas un nouveau nœud représentant c est créé dans un nouveau chemin ainsi qu'un nouvel arc dont le nœud d'origine est le nœud courant avant assignation et dont le nœud cible est le nouveau nœud, également nouveau nœud

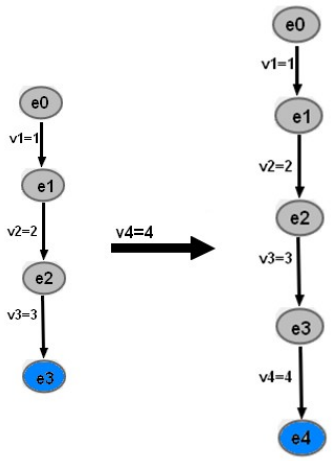


FIGURE III.2 – Assignation : c n'existe pas - Le nœud courant est le dernier de la branche

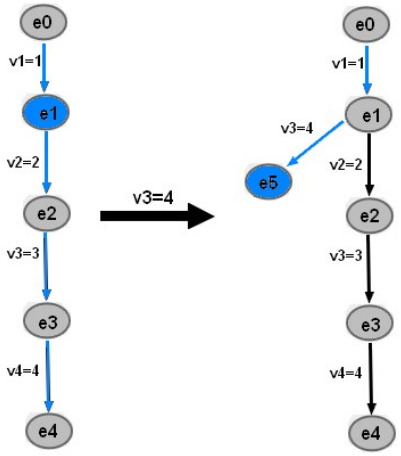


FIGURE III.3 – Assignation : c n'existe pas - Le nœud courant n'est pas le dernier de la branche

courant. Le nouveau chemin créé comporte tous les nœuds prédécesseurs à l'ancien nœud courant.

- c **existe et appartient au chemin courant (Figure III.4)** Ici rien ne change dans la structure, le nœud contenant c devient simplement le nouveau nœud courant. Typiquement ce cas ne se produit que si l'assignation effectuée est un des arcs existants et ayant le nœud courant comme nœud d'origine.
- c **existe mais n'appartient pas au chemin courant (Figure III.5)** Un arc représentant l'assignation ayant le nœud courant comme nœud d'origine et le nœud c comme nœud cible est créé. Le nœud c devient le nœud courant.

Ces cas de figures ne sont valables que si l'assignation est réussie, c'est-à-dire que le résultat respecte le modèle de configuration. Si le moteur de configuration refuse une assignation, celle-ci n'apparaitra ni dans le DAG, ni dans les *history list*. Toutes ces assignations aboutissent à un état où la variable visée possède la valeur spécifiée. Or,

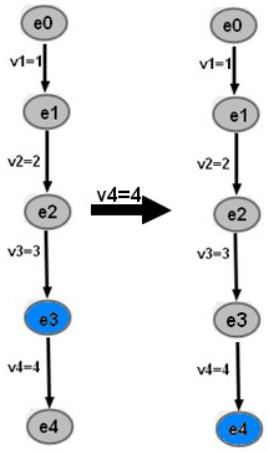


FIGURE III.4 – Assignation : c existe dans le chemin courant

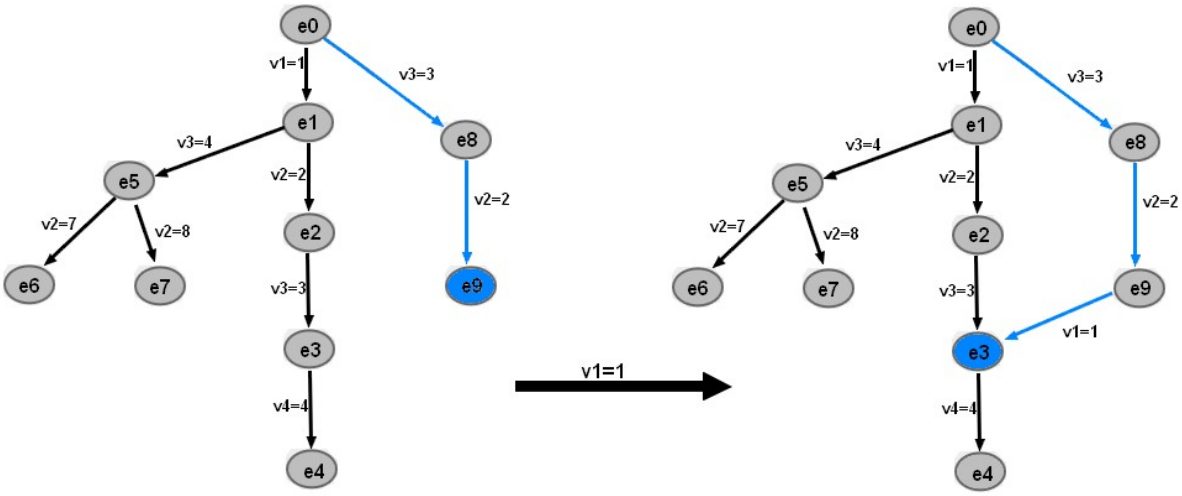


FIGURE III.5 – Assignation : c existe mais pas dans le chemin courant

l'utilisateur peut également décider de rendre indéfinie (nil) la valeur d'une variable ; ce procédé est une seconde commande et s'appelle la *désassignation*. Une désassignation se fait en deux étapes qui sont les suivantes :

- Le nœud précédant l'assignation de la valeur visée dans le chemin courant devient le nœud courant (dans l'exemple à la Figure III.6, le nouveau nœud courant est le nœud racine, précédent l'assignation de v1, la variable visée).
- Les assignations futures (i.e celles suivant l'assignation de la variable visée) sont rejouées dans un autre chemin.

De cette manière l'ancienne branche existe toujours, permettant à l'utilisateur de comparer ses alternatives. Dans les listes, la désassignation prend la forme $d(v, old, new)$ où v est la variable désassignée, *old* est le nœud courant avant exécution de la commande et *new* est le nouveau nœud courant, atteint après exécution. Étant donné que la désassignation se fait en deux étapes et donc au moins deux commandes, celles qui la compose lui est associée dans les *undo* et *redo list*. L'effet de la désassignation est visible sur la Figure III.6. Dans celle-ci, le nœud courant est $e6$ et l'utilisateur décide de désassigner $v1$. Un

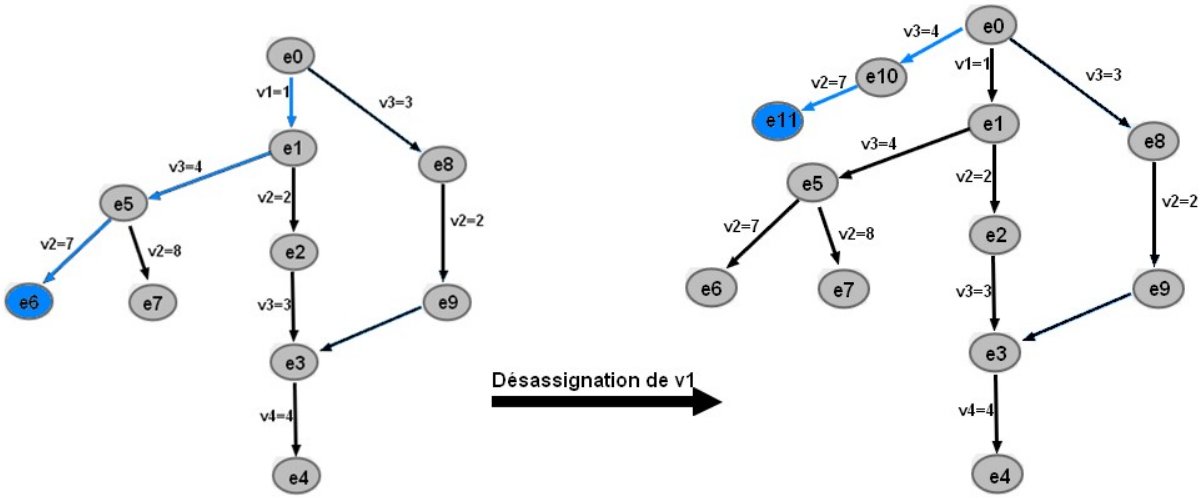


FIGURE III.6 – Exemple de désassignation

retour au nœud $e0$ se produit, suivi du rejeu des assignations $v3 = 4$ et $v2 = 7$ (si le fait de désassigner $v1$ n'a pas provoqué de contrainte empêchant leur ré-exécution). Nous obtenons finalement un nouveau chemin composé de $e0, e10$ et $e11$ avec les variables $v3$ et $v2$ assignées. Comme pour l'assignation, si le moteur de configuration refuse la désassignation, aucun changement ne sera effectué dans les structures.

Précédemment dans cette section, nous avons posé des contraintes sur les assignations, stipulant que chaque variable doit être unique dans un chemin donné afin de prévenir une structure trop complexe du DAG. Cela n'empêche pas pour autant l'utilisateur d'assigner une nouvelle valeur à une variable déjà existante dans le chemin courant. Afin de respecter les contraintes, une commande gère ce cas de figure et ne les enfreint pas, nous l'appelons la *réassignation*. Celle-ci consiste en une désassignation suivie d'une assignation de la nouvelle valeur à la variable visée. Elle est représentée dans l'*undo list* comme $r(v, val, o, t)$ avec v la variable et val la valeur qui les est réassignée, o le nœud courant avant assignation et t le nœud atteint. Nous pouvons observer la réassignation à la Figure III.7. Dans celle-ci, l'utilisateur se situe au nœud $e4$ et réassigne la variable $v3$ à 5. Une désassignation de v3 est dès lors effectuée, provoquant la création d'un nouveau chemin à partir du nœud $e2$. Cette désassignation est suivie de l'assignation $v3 = 5$, le résultat est donc le chemin $(e0, e1, e2, e10, e11)$.

Les contraintes sur les assignations, et par conséquent le système de désassignation / réassignation, ont également été posées afin de prévenir d'éventuels cycles dans le DAG. La désassignation et la réassignation empêchent la création d'un arc qui remonterait dans le DAG et qui créerait un cycle. Pour que ce dernier se forme, il est nécessaire qu'un arc relie un nœud dont la configuration possède moins d'assignations que son nœud d'origine. Or cela n'est pas possible dans notre modèle, en effet pour revenir à un état possédant moins d'assignations, un saut ou une désassignation doit être exécuté, aucun arc ne reliera un tel nœud.

Pour le modèle D.A.C nous avons choisi d'appliquer un modèle d'UNDO linéaire. Cette décision a été motivée par le fait que l'utilisateur peut sauter d'un nœud à l'autre

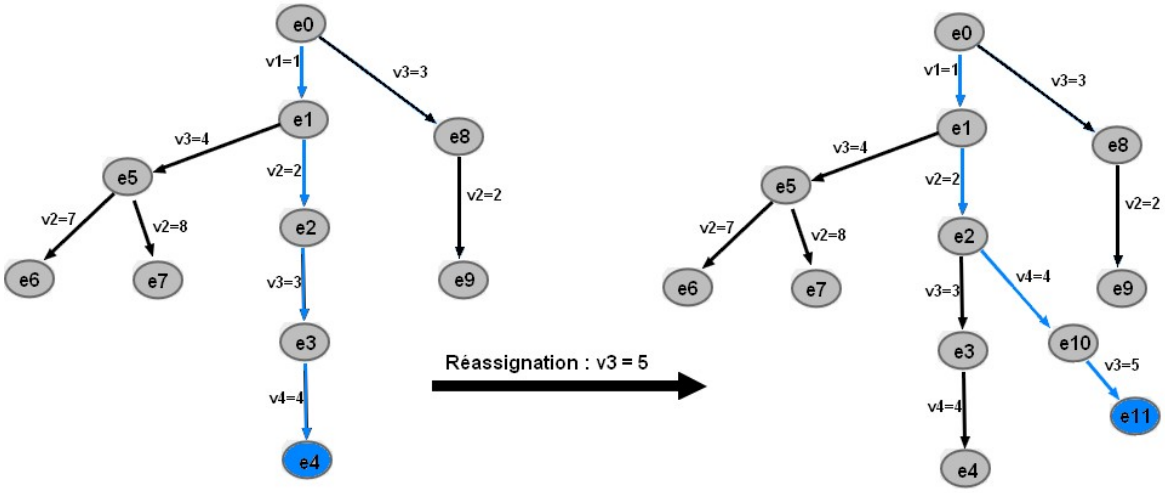


FIGURE III.7 – Exemple de réassignation

dans le DAG à sa guise et désassigner n'importe quelle variable précédemment assignée, le résultat étant le même que si un UNDO sélectif avait été proposé. Nous avons donc choisi de rendre possible l'UNDO pour toute commande exécutée, c'est à dire tous les types d'assignations mais aussi les sauts. Un UNDO est appliqué sur le modèle *dac* (Définition 10) comme $UNDO(dac) = (dac')$ où $dac' = (D', U', R')$ avec l'*undo list* $U' = (c_1, ..., c_{m-1})$, la *redo list* $R' = (c_m, c_n, ..., c_p)$, avec c_m la commande ayant été UNDO, et D' étant le DAG ayant été modifié par l'UNDO de c_m. En effet, toute application d'un UNDO annule l'effet de la commande visée et restitue le DAG tel qu'il était avant l'exécution de celle-ci. Dans la Figure III.8 nous observons une liste de commandes dans l'*undo list* ayant créé le DAG visible dans cette figure. Si nous décidons d'exécuter deux UNDO, les commandes touchées seront un saut ($j(e11, e2)$) et une réassignation (($r(v3, 5, e4)$) dont l'UNDO s'effectuera d'abord sur l'assignation puis la désassignation la composant (la désassignation étant elle-même un saut et une assignation l'UNDO défera : d'abord les 2 assignations et ensuite le saut pour retourner à e4). Le résultat est visible à la Figure III.9. Nous observons dans ce résultat que le DAG correspond à l'état tel qu'il doit être, c'est-à-dire qu'il correspond bien aux commandes présentes dans l'*undo list*. Le saut et la réassignation UNDO ont quant à eux été déplacés dans la *redo list*.

En ce qui concerne le REDO, nous avons choisi de le rendre sélectif. L'utilisateur peut donc choisir n'importe quelle commande de la REDO list et la ré-exécuter. Nous avons $REDO(dac, c)$ où $dac = (D, U, R)$ avec D le dag et U l'*undo list* telle que définie en début de cette section (Définition 10), R est la *redo list* avec $R = (c_n, ..., c, ...c_p)$ et c est la commande à REDO sélectionnée par l'utilisateur. $REDO(dac, c) = (dac')$ où $dac' = (D', U', R')$ avec D' est le DAG ayant été modifié par la ré-exécution de la commande c, $U' = (c_1, ..., c_m, c)$ et $R'(c_n, ..., c_p)$.

Comme nous l'avons fait avec chacun des modèles présentés dans l'état de l'art, nous allons appliquer le modèle D.A.C à l'exemple de substitution 1. Pour rappel, nous souhaitons, à partir d'un enchainement de commandes $(c_1, c_2, c_3, c_4, c_5)$ remplacer c_2 par c'_2 et atteindre $(c_1, c'_2, c_3, c_4, c_5)$. Pour ce faire, nous aurions à UNDO quatre commandes, exécuter c'_2 et REDO les trois autres commandes ajoutées dans la *redo list*, comme dans

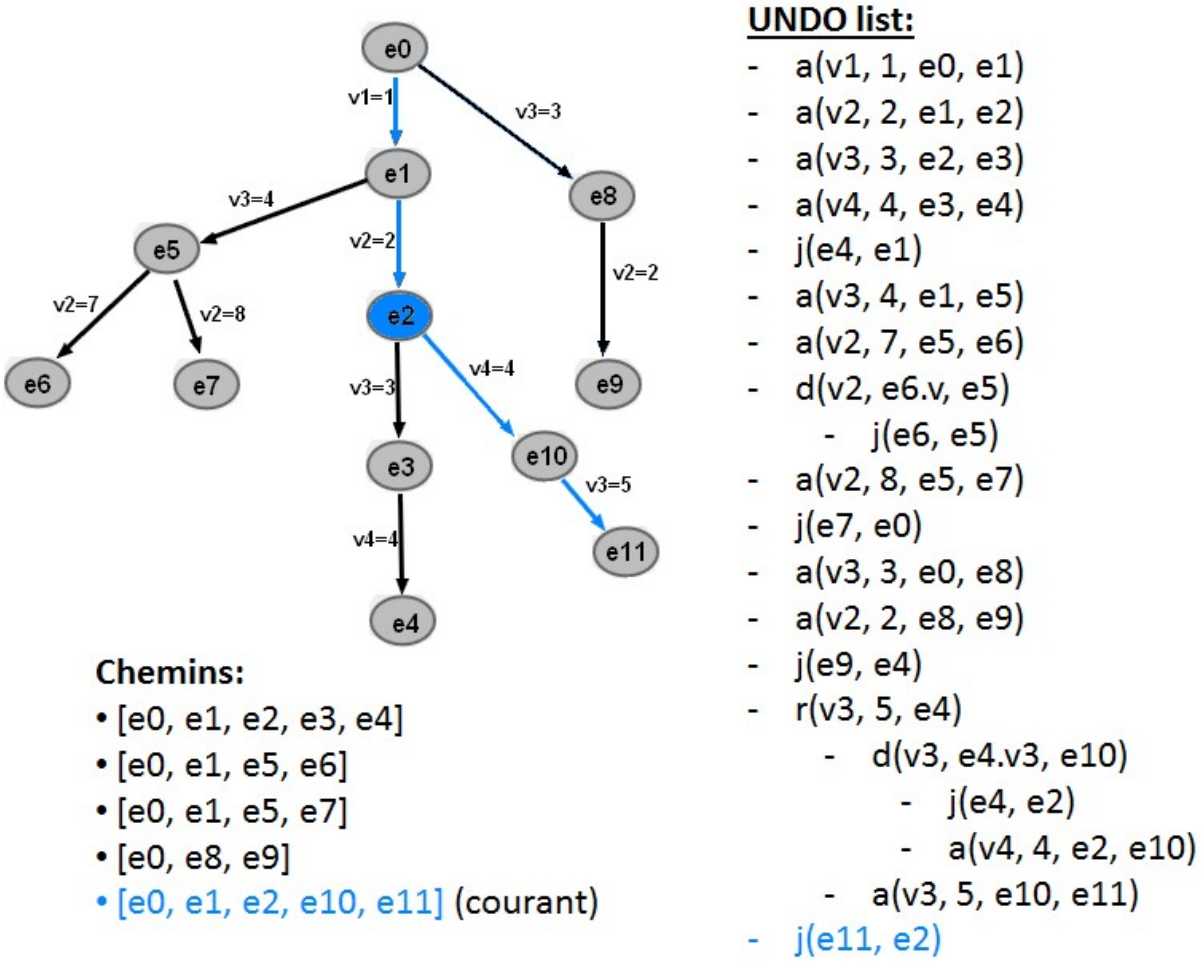


FIGURE III.8 – Modèle D.A.C avant un UNDO

le modèle linéaire. Cela amenant à un total de huit commandes.
Supposons maintenant que toutes les commandes sont des assignations. Nous pourrions simplement effectuer une réassignation de la variable assignée par c_2 et nous obtiendrions un nouveau chemin courant avec $(c_1, c'_2, c_3, c_4, c_5)$ (si le rejeu de c_3, c_4 et c_5 était autorisé par le configurateur). Ce cas de figure, directement lié à la présence du configurateur amène le nombre de commandes nécessaire pour réaliser la substitution à 1.

III.3 Évaluation du modèle par rapport aux critères

1. **Efficacité :** ✓ Grâce au DAG et à la commande "saut", nous permettons à l'utilisateur d'atteindre n'importe quelle configuration précédemment effectuée sans aucun impact sur sa configuration au moment du saut. Toutes les configurations existantes lors de l'exécution le sont toujours après celle-ci et sont toujours accessibles.
2. **Efficience :** ✓ Le système de navigation dans les configurations n'est ici pas linéaire grâce au DAG et à la commande du saut. Cette dernière rend le modèle

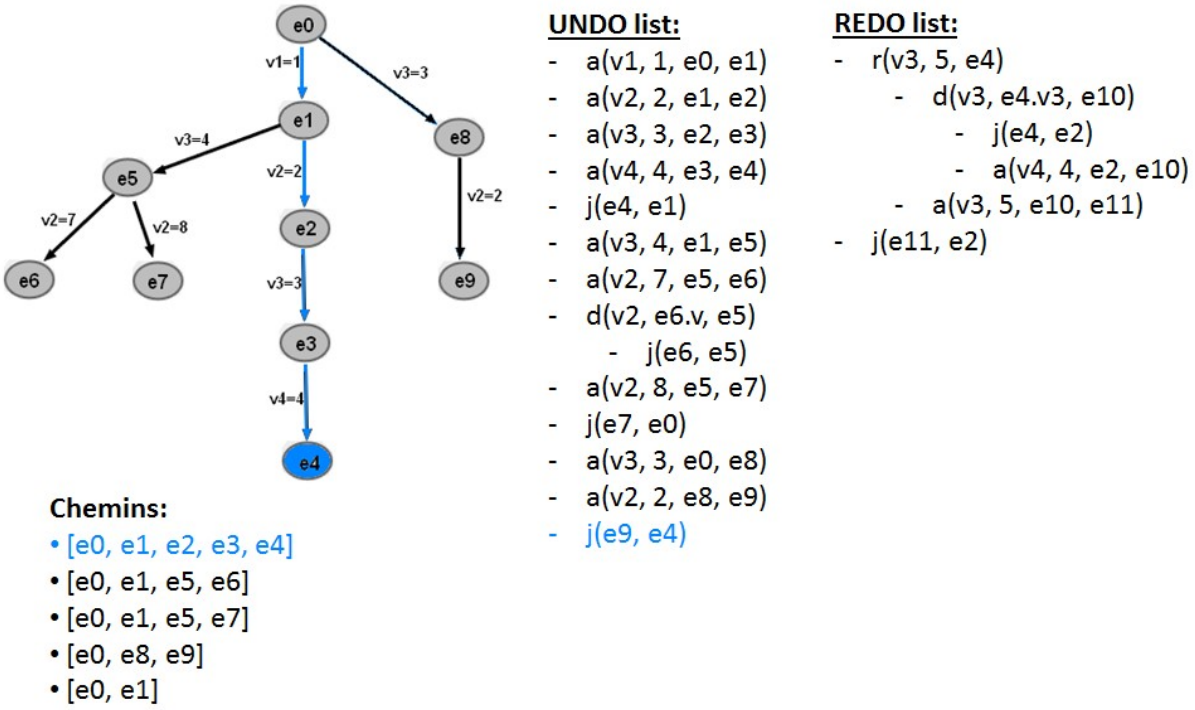
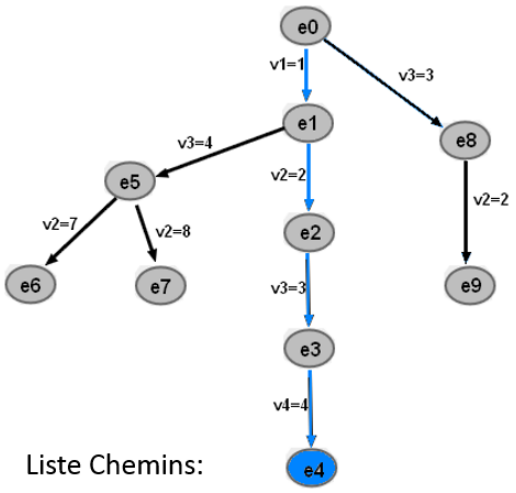


FIGURE III.9 – Modèle D.A.C de la Figure III.8 après deux UNDO

efficient car chaque configuration est atteignable en une seule commande. En plus de cela, nous avions indiqué dans le chapitre précédent que le critère d'efficience était mesuré avec nombre de commandes nécessaires pour réaliser l'exemple de substitution (Section II.1). Si nous nous concentrons sur les history list, le problème est résolu en huit commandes comme la plupart des modèles existants. Cependant, comme nous l'avons vu à la fin de la section précédente, le modèle D.A.C permet grâce à la réassignation de résoudre l'exemple en une seule commande, un meilleur résultat que n'importe quel autre modèle.

3. **Applicabilité à la configuration :** ✓ Ce critère est respecté si le modèle respecte la propriété de la ‘‘*Weakened SEP*” (Propriété 2). Notre système d'UNDO étant linéaire, la partie de la propriété liée à l'UNDO est donc valide (voir Section II.1). De l'autre côté nous avons choisi un REDO sélectif, mettant en danger la propriété puisque n'importe quelle commande faisant partie de la *redo list* peut être REDO. Le respect de la “Weakened SEP” est préservé par le moteur de configuration qui garanti la cohérence des configurations grâce aux contraintes. Une commande dépendant d'une ancienne ne faisant pas partie de la configuration ne pourra être REDO.

Notre modèle respecte donc les critères que nous avions fixés et possède des performances améliorées au niveau de l'efficience par rapport aux autres modèles existants.



Liste Chemins:

- [e0, e1, e2, e3, e4]
- [e0, e1, e5, e6]
- [e0, e1, e5, e7]
- [e0, e8, e9]
- [e0, e1]

UNDO list:

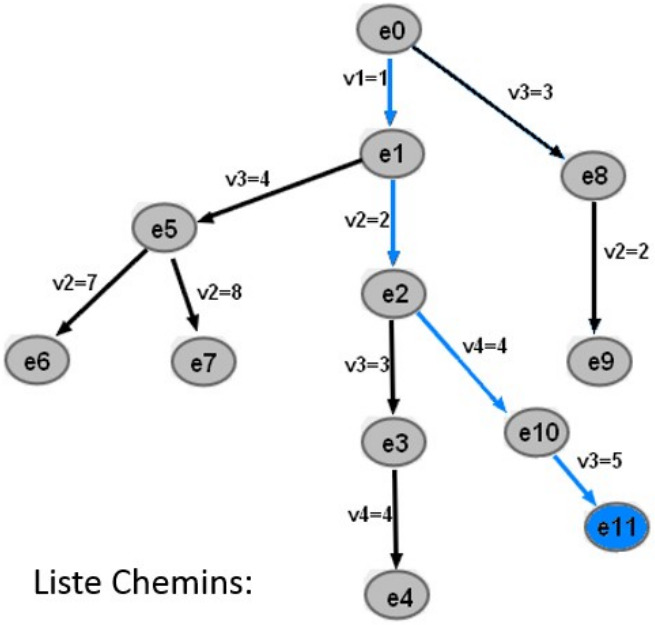
- a(v1, 1, e0, e1)
- a(v2, 2, e1, e2)
- a(v3, 3, e2, e3)
- a(v4, 4, e3, e4)
- j(e4, e1)
- a(v3, 4, e1, e5)
- a(v2, 7, e5, e6)
- d(v2, e6.v, e5)
  - j(e6, e5)
- a(v2, 8, e5, e7)
- j(e7, e0)
- a(v3, 3, e0, e8)
- a(v2, 2, e8, e9)
- j(e9, e4)

REDO list:

- a(v5, 5, e4, e12)
- r(v3, 5, e4) -> REDO souhaité
  - d(v3, e4.v3, e10)
    - j(e4, e2)
    - a(v4, 4, e2, e10)
  - a(v3, 5, e10, e11)
- j(e11, e2)

Prochaine commande : REDO de r(v3, 5, e4)

FIGURE III.10 – Modèle D.A.C avant REDO



Liste Chemins:

- [e0, e1, e2, e3, e4]
- [e0, e1, e5, e6]
- [e0, e1, e5, e7]
- [e0, e8, e9]
- [e0, e1]
- [e0, e1, e2, e10, e11]

UNDO list:

- a(v1, 1, e0, e1)
- a(v2, 2, e1, e2)
- a(v3, 3, e2, e3)
- a(v4, 4, e3, e4)
- j(e4, e1)
- a(v3, 4, e1, e5)
- a(v2, 7, e5, e6)
- d(v2, e6.v, e5)
  - j(e6, e5)
- a(v2, 8, e5, e7)
- j(e7, e0)
- a(v3, 3, e0, e8)
- a(v2, 2, e8, e9)
- j(e9, e4)
- r(v3, 5, e4)
  - d(v3, e4.v3, e10)
    - j(e4, e2)
    - a(v4, 4, e2, e10)
  - a(v3, 5, e10, e11)

REDO list:

FIGURE III.11 – Modèle D.A.C après REDO

Chapitre IV

Implémentation - Back End

Afin de pouvoir fournir un logiciel exploitant notre modèle D.A.C lors d'un processus de configuration et d'évaluer ce modèle par des études de cas concrètes, nous avons implémenté une Application Programming Interface (API) fournissant les fonctionnalités de notre modèle. Dans ce chapitre, nous présenterons en premier lieu l'API publique qui nous a été fournie et que nous avons utilisé pour le développement. Ensuite, nous parlerons de notre implémentation du modèle D.A.C.

IV.1 Structure principale

Le modèle D.A.C peut être séparé en quatre parties distinctes : la structure du DAG, les *history lists*, les commandes et l'accès aux structures. Le package du DAG est composé du graphe lui-même ainsi que des éléments qui le composent, c'est-à-dire les chemins, les arcs et les nœuds. Le package des *history lists* contient l'*undo list* et la *redo list*. Celui des commandes reprend toutes celles citées dans le chapitre précédent, c'est-à-dire les assignations, désassignations, réassignations et saut. Enfin le dernier des packages fournit un accès aux structures ainsi qu'à l'exécution de commandes. La structure est montrée à la Figure IV.8. En plus de ces composants, nous nous servons de l'API publique développée pour le moteur de configuration.

IV.2 CONFETOOLS

Nous avons utilisé suite d'outils nommée CONFETOOLS [5] qui a été développée par la startup CONF&TI. CONFETOOLS permet de spécifier une ligne de produits et d'implémenter, de tester et de déployer une interface de configuration à partir de cette ligne de produits [5]. Comme expliqué dans l'introduction, CONFETOOLS est séparé en deux parties : la partie modélisation de la ligne de produits et la partie logicielle.
La modélisation se fait grâce à un langage développé par la startup permettant d'exprimer les composants, leurs cardinalités, leurs hiérarchies ainsi que leurs attributs de façon claire (i.e "*human readable*"). Ce langage est appelé TVL (Textual Variability Language), sa syntaxe et sémantique sont décrits dans le document [9]. Un exemple de document TVL est également montré en Annexe A.
La seconde partie de la suite CONFETOOLS reprend tout ce qui est nécessaire à la génération de la partie IHM du configurateur côté client et à la gestion de la configu-

ration côté serveur. La présentation du modèle TVL dans l'interface est définie grâce à deux autres langages : TVDL et FCSS [5]. Le premier définit des vues (listes) sur base des composants et attributs du modèles TVL tandis que le second spécifie la façon dont seront agencés ces composants dans l'interface.

Une fois le configurateur généré, une application côté serveur gère la configuration et la gestion des contraintes.Cette partie de CONFETOOLS a été utilisée afin d'implémenter notre modèle côté back-end. Cette API est divisée en deux packages. Le premier package nommé "*domain*" (Figure IV.1) contient les concepts d'options de configuration (variables) et de valeurs associées à ces options. Une valeur peut être soit *null*, booléenne ou sous forme de chaine de caractères. Le concept d'assignation d'une option à une valeur fait également partie du package. Dépendant du résultat d'une telle assignation, nous trouvons deux objets représentant soit le cas d'une réussite, soit le cas d'un échec. Chaque objet symbolisant un résultat contient l'assignation et les propagations en résultant. Celles-ci sont les conséquences provoquées par l'assignation a. Une propagation sera par exemple une assignation b forcée résultant d'une valeur associée à une variable dépendant de a. Enfin nous trouvons une implémentation du pattern Memento, utilisé pour enregistrer l'état du configurateur à un instant donné.

L'API possède un autre package appelé "*application*" fournissant des services de configuration. Un schéma de ce package est visible à la Figure IV.2. Tout d'abord nous avons l'interface *ConfigOptionFinder* offrant la possibilité de retrouver une option dans une configuration sur base de son nom ou de son *id*. Nous retrouvons aussi l'interface *ConfigurationReader* fournissant des méthodes pour récupérer la configuration courante ou une option dans une configuration sur base de son nom. Nous avons également l'interface *Formula* permettant de récupérer la valeur d'une option dans une configuration. Enfin nous trouvons l'interface *Configurator*, permettant à l'utilisateur d'utiliser les services du configurateur. Ceux-ci incluent l'initialisation d'une configuration, l'assignation ou la désassignation d'une option, l'ajout ou le retrait d'une option dans la configuration, la sauvegarde et la restauration d'un état de configuration grâce au *ConfiguratorMemento* et enfin la récupération de la configuration courante.

IV.3 L'implémentation

Pour ce qui est de la structure du projet nous avons décidé de fournir une classe offrant l'accès aux structures principales (le DAG et les listes), permettant aux commandes d'accéder aisément à celles-ci. Les méta-commandes agissent sur les commandes. Ces dernières ont chacune leurs façon de s'exécuter mais chacune d'entre elles affecte les structures. Nous distinguons donc une séparation entre les structures, les commandes et l'accès à celles-ci.

IV.3.1 Le Modèle D.A.C

Les structures sont séparées en deux packages différents, un pour le DAG et un pour les listes. Dans le DAG nous retrouvons quatre classes, chacune d'elle représentant un

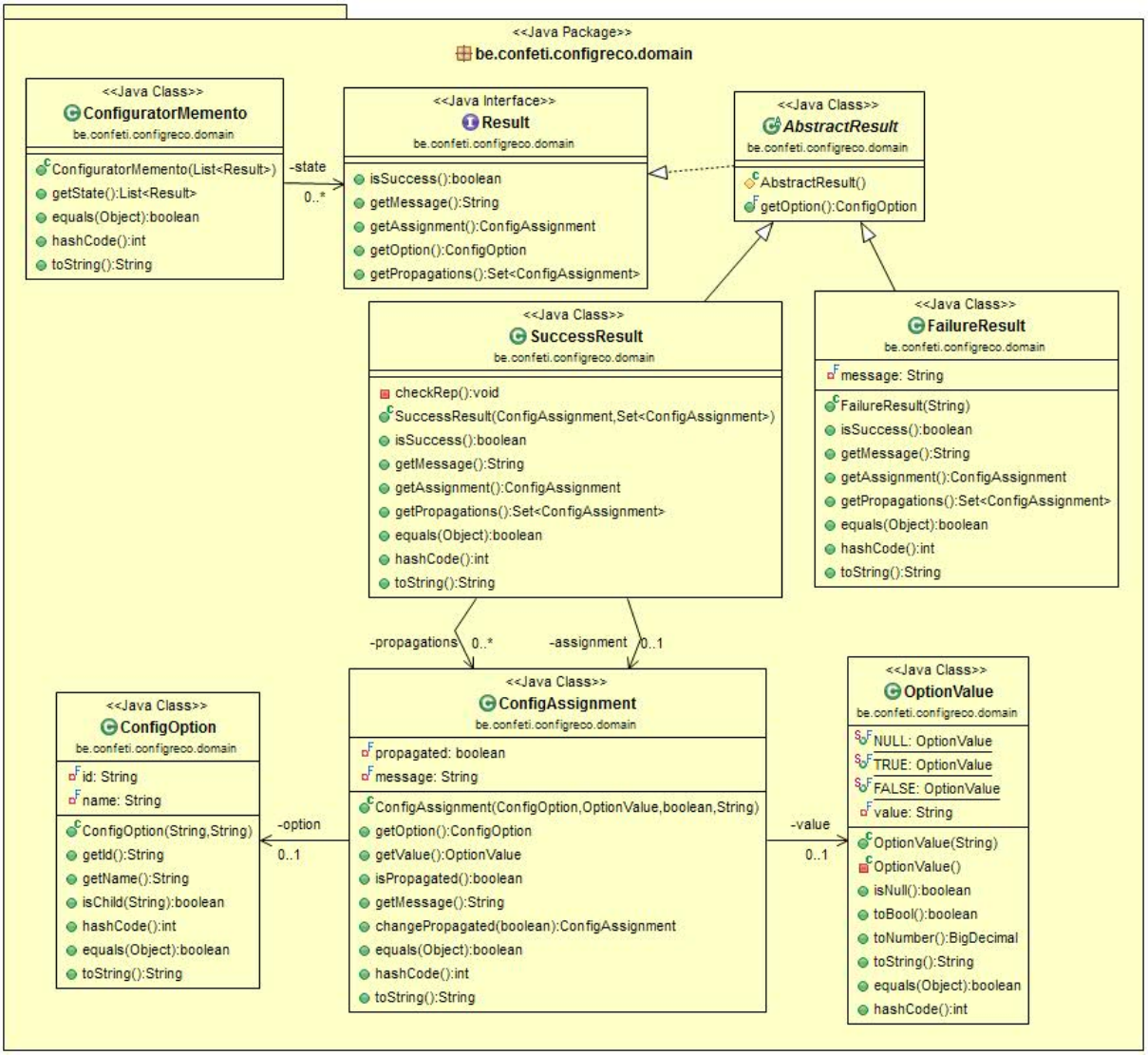


FIGURE IV.1 – Package "*domain*" de l'API Publique

composant du graphe. À ce titre nous retrouvons la classe *Node* pour le nœud, la classe *Arc*, la classe *Path* pour les chemins ainsi que la classe *DAG* symbolisant le graphe lui-même. Le diagramme de classe de ce package est montré en Figure IV.3.

Dans la classe *Node*, nous utilisons un *ConfiguratorMemento* pour représenter l'état de la configuration au moment de la création du nœud.
L'objet *Arc* possède un *Node origin*, un *Node target* et un *ConfigAssignment* représentant l'assignation qui lui est associée et qui depuis la configuration du nœud *origin* mène à celle du nœud *target*.
La classe *Path* symbolise un chemin et possède une liste d'Arc. En fonction des besoins requis par les commandes telles que les assignations, cette classe possède deux méthodes parcourant la liste des arcs. Tout d'abord nous avons la méthode *checkVariableExistence* (Listing IV.1) qui consiste simplement à parcourir le chemin à la recherche du noeud précédant l'assignation d'une option de configuration donnée.

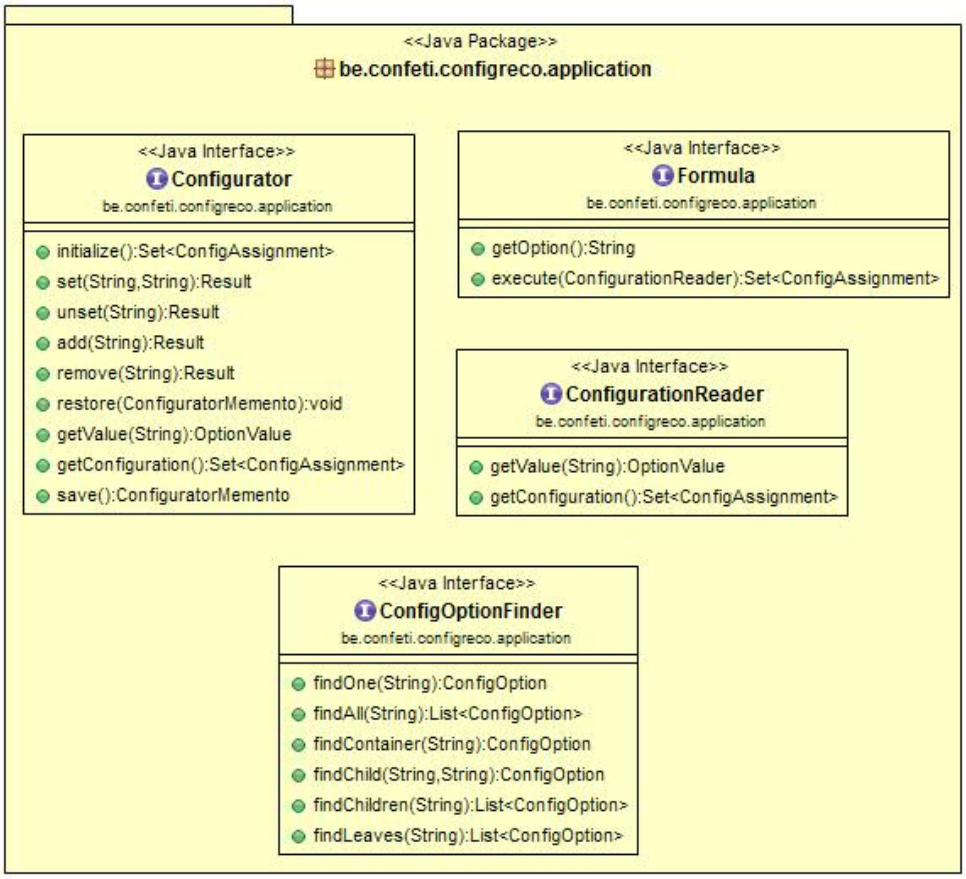


FIGURE IV.2 – Package "*application*" de l'API Publique

```
1 public Node checkVariableExistence(ConfigOption co){
2    for all a in arcs do
3       if a.assignation.option.name == co.name then
4          return a.originNode
5       end if
6    end for
7
8    return null
9 }
```

Listing IV.1 – Méthode *checkVariableExistence* de la classe *Path*

L'autre méthode s'intitule *checkNodeExistence* (Pseudo code dans le Listing IV.2) et vérifie dans un chemin l'existence d'un nœud donné. Dans le chemin courant, le nœud recherché ne peut se situer avant le nœud courant (contrainte posée dans la Section III.2). C'est donc uniquement à partir de celui-ci que chaque nœud du chemin est comparé au nouveau (d'où l'utilisation d'une variable *currentNodeFound*, placée à *true* si la condition à la ligne 14 est respectée). Pour des raisons de performances, nous avons décidé de vérifier, avant chaque comparaison du contenu des nœuds, si le nombre d'assignations dans leurs configurations respectives est égal (ligne 6), et cela peu importe le chemin dans lequel nous sommes. Cela évite de comparer inutilement toutes les configurations du chemin. Si le nombre d'assignation d'un noeud du chemin est inférieur à celui du noeud

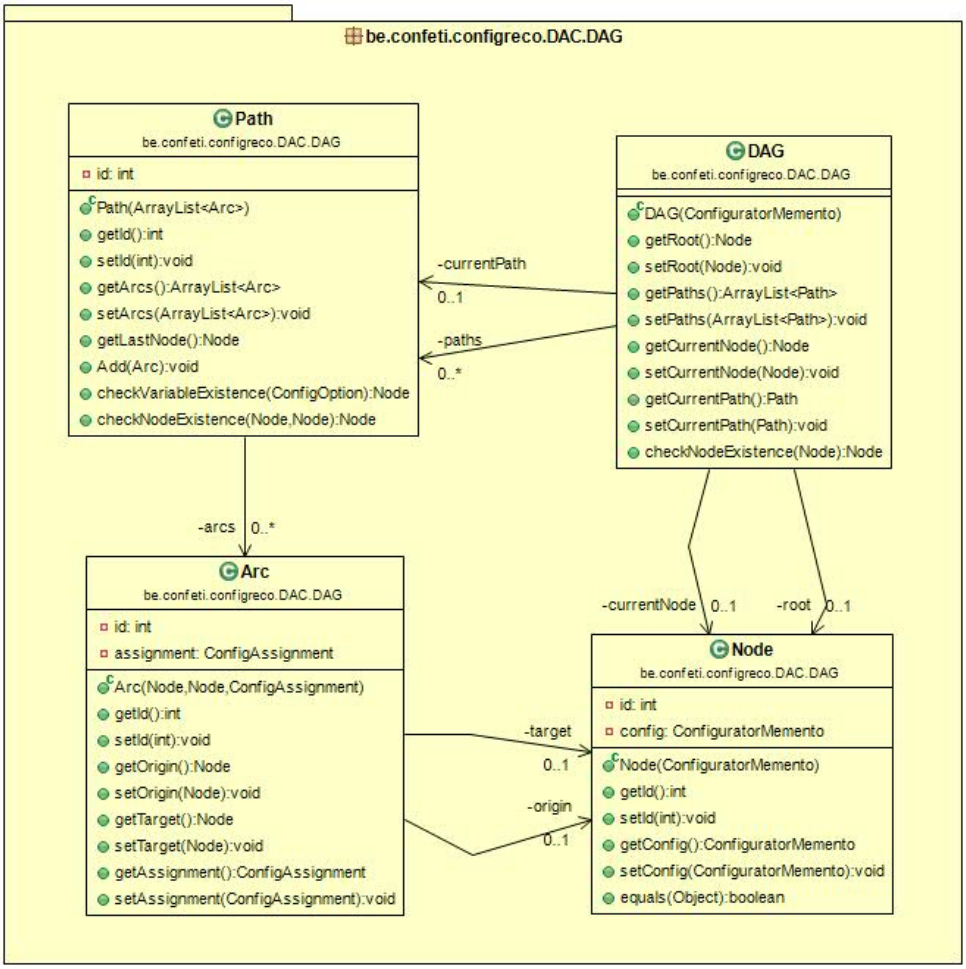


FIGURE IV.3 – Package "*DAG*"

nouvellement créé, nous continuons à progresser dans le chemin. S'il est supérieur cela signifie que le nœud recherché ne fait pas partie du chemin, le nombre d'assignations dans les nœuds d'un même chemin ne pouvant qu'augmenter. Si le nombre d'assignations est égal, le contenu des nœuds est comparé et si eux-mêmes sont égaux, le nœud est récupéré.

```
1  public Node checkNodeExistence(Node current, Node toCheck, boolean
       currentPath){
2     currentNodeFound := false
3
4     for all a in arcs do
5        if currentNodeFound then
6           if a.targetNode.list.size == nodeToCheck.list.size then
7              if a.targetNode == nodeToCheck then
8                 return a.targetNode
9              end if
10          else if a.targetNode.list.size > nodeToCheck.list.size then
11             return null
12          end if
13       else if
```

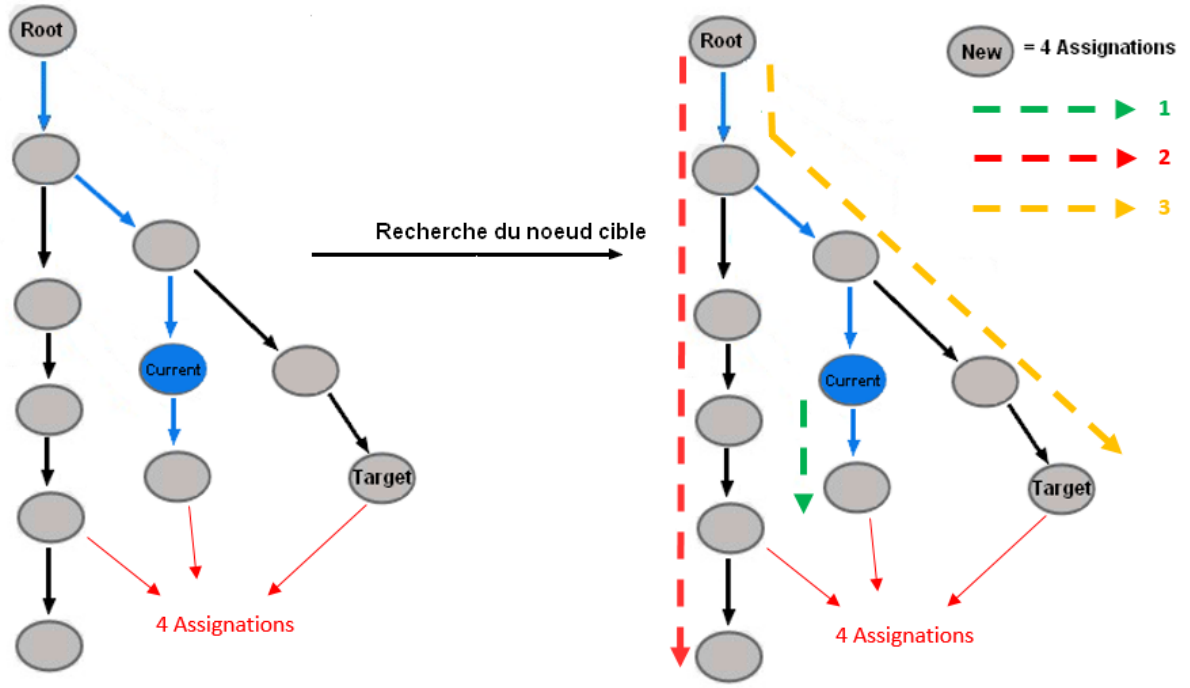


FIGURE IV.4 – Recherche d'un nœud existant

```
14          if a.originNode.id == currentNode.id then
15             currentNodeFound := true
16          end if
17       end if
18    end for
19 }
```

Listing IV.2 – Méthode *checkNodeExistence* de la classe *Path*

Par exemple, dans la Figure IV.4, nous avons dans la partie gauche, un DAG avec trois chemins différents. Le chemin et le nœud courant sont surlignés en bleu et le nœud cible est appelé "Target". Le nouveau nœud créé par une nouvelle assignation possède une configuration en contenant quatre. Dans le DAG, nous avons indiqué un nœud dans chaque chemin possédant quatre assignations. Selon l'algorithme, le contenu de ces noeuds-là sera comparé avec le nouveau nœud tandis que les autres noeuds ne subirons qu'une simple comparaison du nombre d'assignations dans leurs configurations. Dans la partie droite, nous voyons le même DAG mais avec des indications quand à l'ordre dans lequel va se dérouler la recherche du nœud existant. Cela partira d'abord du chemin courant et plus précisément du nœud courant. Le nœud le suivant dans le chemin va ensuite être comparé car le nombre d'assignations est identique au nœud "New". Ces deux nœuds n'étant, en revanche, pas identiques, la cible ne fait pas partie de chemin. L'algorithme va donc parcourir les autres chemins dans l'ordre dans lequel ils ont été créés. Dans chacun d'eux, un seul nœud va être comparé à "New". Une fois le noeud "Target" trouvé, celui-ci est récupéré et un nouvel arc sera créé avec "Current" comme noeud d'origine et "Target" comme noeud cible.

Enfin nous arrivons à la fin de ce package *DAG* et nous retrouvons la classe éponyme.

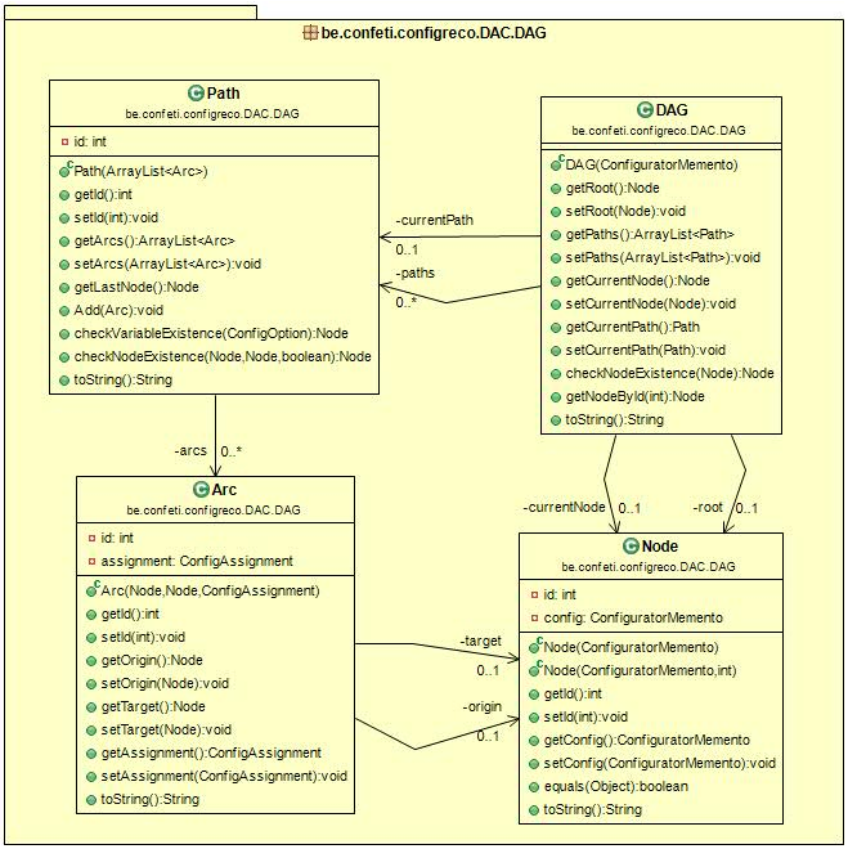


FIGURE IV.5 – Package "*History*"

Celle-ci est composée d'un *Node* correspondant au nœud racine, d'une liste de *Path* et de deux autres variables indiquant le chemin et nœud courants. Une méthode *checkNodeExistence* est également présente ici et c'est cette méthode qui définit l'ordre de parcours des chemins du DAG (comme l'ordre de la Figure IV.4). Cela commencera toujours par le chemin courant suivit des autres chemins, dans l'ordre dans lequel ils ont été créés. Lors du parcours des chemins, leur méthode *checkNodeExistence* est appelée et le noeud à vérifier leur est fourni.

Le second package contenant des structures est nommé *history* et contient deux classes : *UndoList* et *RedoList*, visibles à la Figure IV.5. . Chacune possède une liste d'objet *Command* et des méthodes pour y accéder et pour les modifier (par exemple en ajoutant ou supprimant des commandes de la liste). Les deux classes permettent de récupérer la prochaine commande à UNDO ou REDO respectivement grâce aux méthodes *getCommandToUndo* ou *getCommandToRedo*. Par rapport à la classe *UndoList*, la classe *RedoList* permet de sélectionner quelle commande dans la liste doit être REDO et elle offre également une méthode pour effacer la liste. Cette dernière est appelée lors de l'exécution d'un *Jump*.

IV.3.2 Les Commandes

Le package *Commands* comporte les classes représentant les commandes que va exécuter l'utilisateur. Nous y trouvons deux classes abstraites : *Command* et *MetaCommand*.

Quatre classes héritent de la première, à savoir : *Assignment*, *Desassignment*, *Reassignment* et *Jump* et deux héritent de la seconde : *Undo* et *Redo*. La classe abstraite *Command* définit deux méthodes abstraites : *execute* qui, une fois implémentée, procédera à l'exécution de la commande et *toUndo* qui renverra l'objet *Command* à UNDO. Comme dans le chapitre précédent, commençons par le *Jump*. Celui-ci possède comme variables un *Node* origin et un *Node* target. Nous avons également ajouté un booléen appelé *withinDorR* qui est indiqué vrai si le saut est exécuté dans le cadre d'une désassignation ou d'une réassignation ; si tel est le cas, le saut ne doit pas être ajouté à l'*undo list*. Quand le *Jump* est exécuté (Listing IV.3), l'état du noeud cible est restauré grâce au *Memento*. Le nœud cible est indiqué comme nœud courant et la commande est ajoutée à l'*undo list* si elle n'est pas exécutée pendant une désassignation ou une réassignation.

```
1  public void execute(){
2     Configurator.restore(targetNode.config)
3
4     DAC.currentNode := target
5     if !withinDesassign then
6        DAC.UndoList.add(this)
7     end if
8
9     DAC.RedoList.clear()
10 }
```

Listing IV.3 – Exécution d'un Jump

Passons maintenant à l'objet *Assignment*, créant un nouvel arc et dont nous en retrouvons les mêmes caractéristiques : deux *Node*, un origine et un cible ainsi que l'assignation en question, en l'occurrence ici, un objet *Result*. Comme dans le *Jump* nous avons un booléen indiquant si la commande est exécutée dans le cadre d'une réassignation. La méthode d'exécution de cet objet est représenté par les Listings IV.4, IV.5 et IV.6 Avant tout, il est utile de détecter si nous sommes dans un cas d'assignation ou de réassignation (Listing IV.4). Cette première vérification se fait grâce à l'API Confetools (Section IV.2), celle-ci recherche si l'option assignée existe déjà dans la configuration. Le cas échéant, nous sommes en présence d'une réassignation.

```
1  public void execute() {
2     configOption := API.ConfigOptionFinder.find(option)
3     if co != null then
4        //Nous sommes dans une assignation
5        r := new Reassignment(this)
6        r.execute
7     else
8     //Nous sommes dans une assignation
9     end if
```

Listing IV.4 – Vérification du type de la commande : Assignation ou Réassignation

Si nous sommes dans une assignation, la prochaine étape est de créer un nouveau noeud en enregistrant, grâce au Memento, la nouvelle configuration créée par le configurateur. Il est ensuite utile de vérifier si celle-ci n'existe pas déjà quelque part dans le DAG (Listing IV.5). Pour ce faire, une fois le noeud créé nous faisons appel au DAG afin qu'il parcourt ses chemins et vérifie l'existence du nouveau noeud (comme dans la Figure IV.4).

```
1  //Memento
2  newNode := new Node(Configurator.save());
3
4  //Check if this Node already exists
5  alreadyExists = DAG.checkNodeExistence(newNode);
6  if alreadyExists != null then
7     newNode := alreadyExists;
8  end if
9
10 //Create a new arc that contains the assignment
11 newArc := new Arc(origin, newNode, assignment);
```

Listing IV.5 – Vérification de l'existence du noeud et création d'un nouvel Arc (suite de la méthode du Listing IV.4)

A présent, nous souhaitons savoir si ce nouveau noeud peut-être ajouté à la suite du chemin courant ou s'il est nécessaire de créer un nouveau chemin et de l'y ajouter. Nous devons donc vérifier si le noeud courant actuel possède des successeurs (Listing IV.6, ligne 7). Si c'est le cas nous devons créer un nouveau chemin, y ajouter tous les noeuds du chemin courant jusqu'au noeud courant et y ajouter le nouveau noeud (lignes 10-18). S'il ne possède pas de successeurs, nous l'ajoutons au chemin courant (ligne 8). Le noeud courant est mis à jour. Si un nouveau chemin a été créé, il sera ajouté au DAG comme nouveau chemin courant. La commande d'assignation est ensuite ajoutée à l'*undo list* si elle n'est pas exécutée au sein d'une désassignation ou d'une réassignation.

```
1  if DAG.currentPath == null
2     //It's the first node
3     ArrayList<Arc> arcs = new ArrayList<Arc>();
4     arcs.add(newArc);
5     DAG.setCurrentPath(new Path(arcs));
6  else
7     if DAG.currentPath.lastNode == DAG.currentNode then
8        DAG.currentPath.add(newArc)
9     else
10       newPath := new Path
11       //This path contains every node until origin
12       for all a in DAG.currentPath.arcs do
13          newPath.arcs.add(a);
14          if a.target. == DAG.currentNode then
15             //Once origin achieved, new arc is added
16             newPath.arcs.add(newArc);
17             break;
```

```
18            end if
19         end for
20
21         DAG.add(newPath);
22         DAG.currentPath := newPath
23      end if
24   end if
25
26   DAG.currentNode := newNode
27
28   if !withinDesassign then
29      DAC.UndoList.add(this);
30   end if
31}
```

Listing IV.6 – Ajout d'un nouveau noeud au DAG (suite du Listing IV.5)

Si après la partie montrée par le Listing IV.4, nous tombons dans un cas de réassignation, celle-ci est réalisée en désassignant la valeur d'origine de l'option visée puis en lui assignant une nouvelle valeur.

La désassignation est exécutée en deux parties : la première partie est montrée dans le Listing IV.7. D'abord nous exécutons le saut jusqu'au noeud précédant l'assignation (ligne 16), appelons ce noeud b. Ensuite, si la configuration résultant de la désassignation est la même que celle du noeud b, il n'y a pas besoin de rejouer quoi que ce soit. Sinon nous procédons à la création d'un nouveau chemin dans lequel nous ajoutons les noeuds précédant b. Puis nous récupérons les assignations suivants le noeud courant dans le chemin courant afin de les rejouer plus tard.

```
1  ArrayList<Assignment> toUndo = new ArrayList<Assignment>();
2  //Jump j was initiated when this was
3  public void execute() {
4     ArrayList<ConfigAssignment> toReplay = new ArrayList<ConfigAssignment>();
5     Path newPath := null;
6     Path currentPath := DAG.currentPath;
7     Node beforeAssignment :=
          currentPath.checkVariableExistence(this.desassignment.option);
8     ConfiguratorMemento isJumpTarget = Configurator.save();
9
10    //occurs when redo
11    if j == null then
12       //set the target node and indicates that it is executed within a
             desassignment
13       j := new Jump(beforeAssignment, true);
14    endif
15    //Execute Jump
16    j.execute();
17
18    //compare configurator state and target state
```

```
19    if !j.target.config.state.equals(isJumpTarget) then
20       boolean add := false;
21       boolean first := false;
22       //Get assignments to replay
23       for all a in DAG.currentPatharcs do
24          if a.assignment.option.name.equals(this.desassignment.option.name) then
25             // we reached the currentNode, we can add the next assignment to
                  the replay list
26             add = true ;
27          endif
28
29          if add then
30             if first then
31                toReplay.add(a.assignment);
32             else
33                first = true;
34             end if
35          else
36             arcsInNewPath.add(a);
37             //arcs to Undo when needed
38          end if
39       end for
```

Listing IV.7 – Exécution d'une désassignation

La seconde partie de l'exécution est le rejeu des assignations (Listing IV.7). Celle-ci parcours la liste des assignations qui doivent être rejouées et les ajoute au nouveau chemin. Nous terminons par ajouter le nouveau chemin au DAG et à l'indiquer comme nouveau chemin courant.

```
1     //if things have to be replayed
2     //get these assignments in list and execute them without adding them in
          undo list as command
3     if !testReplay.isEmpty() then
4     for all ca in toReplay do
5        Result result := Configurator.set(a.option, a.value);
6        if result.isSuccess() then
7           //Create new Node
8           Node target := new Node(Configurator().save());
9           //Get origin node
10          Node origin := toReplay.get(0)==ca ? j.target :
                arcsInNewPath.last.target;
11          //Add new arc
12          arcsInNewPath.add(new Arc(origin,target,result.assignment));
13
14          Assignment toAddtoUndo := new Assignment(result, origin);
15          toAddtoUndo.setTarget(target);
16          toUndo.add(toAddtoUndo);
17          DAG.setCurrentNode(target);
18       end if
19    end for
```

```
20    end if
21
22    newPath := new Path(arcsInNewPath);
23    DAG.add(newPath);
24    DAG.currentPath := newPath;
25
26    else
27       //No replay, just a jump
28    end if
29
30    if !withinReassignment then
31       DAC.UndoList.add(this);
32    end if
33 }
```

Listing IV.8 – Exécution d'une désassignation (suite du Listing IV.7)

Au début de cette section nous parlions d'une seconde classe abstraite : *MetaCommand*, possédant une seule méthode abstraite : *execute*. Deux classes héritent de *MetaCommand* : *Undo* et *Redo*. Chacune possède une variable de type *Command* appelée *toUndo* ou *toRedo* selon la classe dans laquelle nous sommes. Dans le cas du *Redo*, la variable est initialisée dans la méthode *execute* en récupérant la *Command* depuis la *redo list*. Si la commande récupérée est *null*, cela siginifie que la liste est vide et donc rien ne se passe. En revanche, si nous récupérons bien une commande, celle-ci est exécutée, retirée de la *redo list* puis ajoutée à l'*undo list* (Listing IV.9). Si la commande est une désassignation nous ré-exécutons d'abord le saut puis la liste des assignations liées. Si c'est une simple assignation, nous devons utiliser le configurateur pour de nouveau assigner la valeur à la variable. Nous indiquons ensuite que le noeud origine de cette assignation est le noeud courant et nous exécutons la commande. Dans les autres cas (réassignation ou saut), aucune action supplémentaire n'était nécessaire, nous exécutons donc simplement la commande.

```
1  public void execute(){
2     toRedo := DAC.redoList.getCommandToRedo()
3
4     //If null, redo list is empty
5     if toRedo != null
6        if toRedo instanceof Desassignment then
7           toRedo.jump.execute();
8
9           //toRedo.toUndo => Assignment list to replay in a Desassignment
10          for all a in toRedo.toUndo do
11             a.execute();
12          end for
13       else
14       if toRedo instanceof Assignment then
15          Configurator.set(toRedo.option.name, toRedo.value);
16
17          toRedo.origin(DAG.currentNode);
```

```
18          toRedo.setWithinDesassign(true);
19          toRedo.execute();
20       else
21          toRedo.execute();
22       end if
23
24       DAC.redoList.remove(toRedo);
25    else
26       //Redo list is empty
27    end if
28 }
```

Listing IV.9 – REDO

En ce qui concerne l'UNDO, la méthode *execute* récupère également la commande depuis l'*undo list* et assigne *toUndo* puis exécute la méthode *undo* sur cette variable. Quatre cas de figure sont à présent possibles en fonction du type de commande *UNDO*.

- ***toUndo* est un *Assignment* :** le nœud courant dans le DAG ainsi que l'arc le possédant comme nœud cible sont supprimés du DAG et le nœud origine de cet arc est le nouveau nœud courant. Lorsqu'un *Assignment* est UNDO, le nœud concerné dans le DAG est toujours le dernier du chemin. En effet, l'UNDO étant linéaire dans notre modèle, l'*Assignment* renvoyé pour UNDO est toujours le dernier de la liste, correspondant toujours au dernier du chemin courant.
- ***toUndo* est un *Desassignment* :** la liste des *Assignment* associés est récupérée. Si elle n'est pas vide, tous les *Assignment* en faisant partie sont UNDO. La commande *Jump* associée est ensuite UNDO.
- ***toUndo* est un *Reassignment* :** les deux variables sont récupérées (*Assignment et Desassignment*) puis UNDO.
- ***toUndo* est un *Jump* :** la méthode *toUndo* de l'objet *Jump* est appelée, celle-ci le renvoie avec le nœud origine et le nœud cible inversés pour ensuite être exécuté.

Une fois l'UNDO de la commande effectué, celle-ci est retirée de l'*undo list* puis ajoutée à la *redo list*.
Une représentation du package *Commands* est visible à la Figure IV.6. On peut y observer les liens entre les classes dont ceux d'héritage des objets *Command* et *MetaCommand* avec leurs superclasses.

IV.3.3 Accès

Le package *dacAccess* est visible à la Figure IV.7. Dans celui-ci nous retrouvons d'abord une classe *Access* dont le but est d'initialiser les structures, de les mettre à jour après l'exécution d'une commande ainsi que de fournir à l'extérieur la possibilité d'appeler une des quatre commandes possibles. La méthode *assign* par exemple, prend en paramètre un objet *SuccessResult* et exécute une commande *Assignment*. S'en suit l'exécution de la méthode *update* mettant à jour les structures (Listing IV.10).

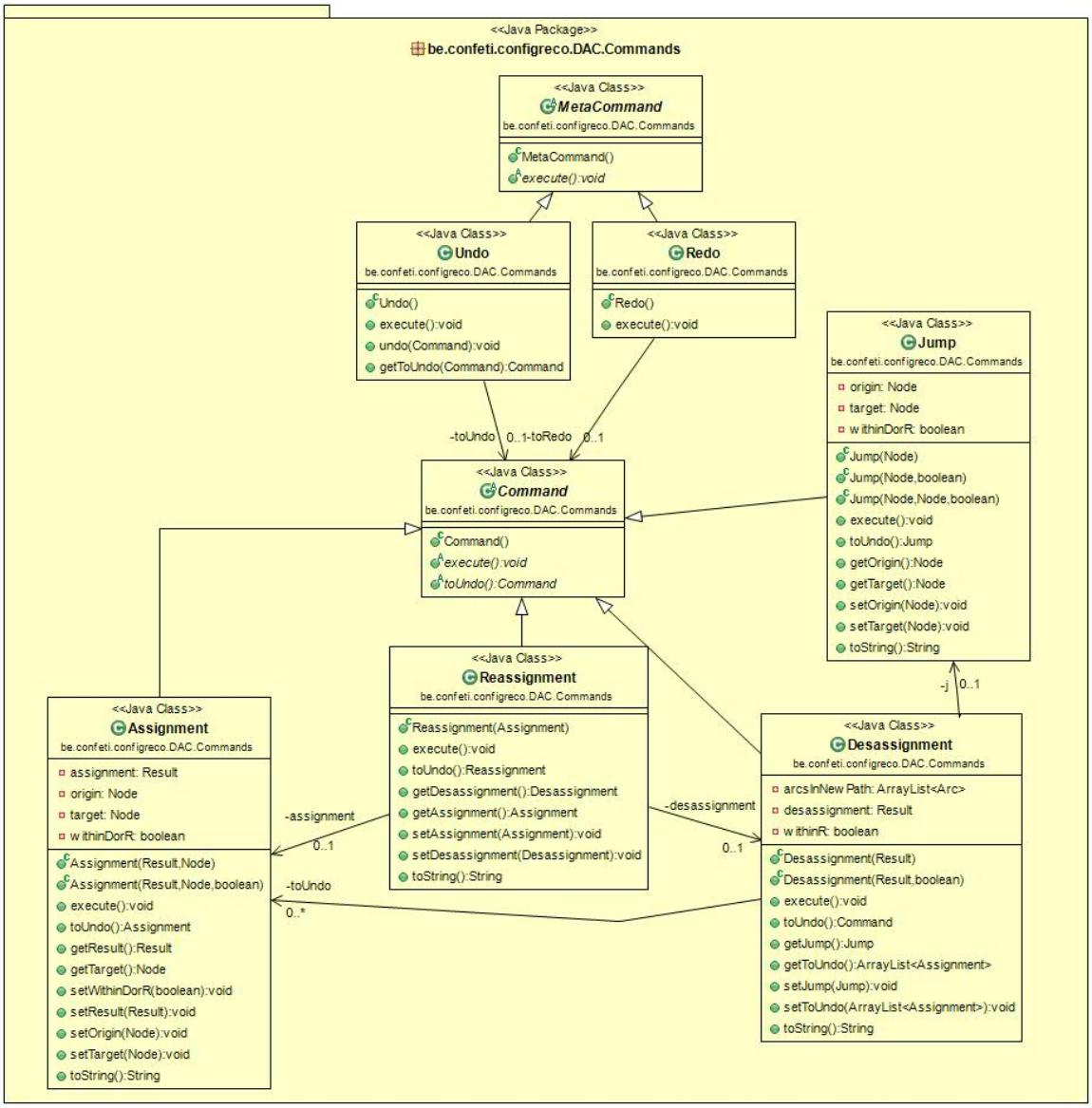


FIGURE IV.6 – Package "*Commands*" de l'API Publique

```
1 public void assign(SuccessResult r){
2 Command c := new Assignment(r, DAG.currentNode);
3 c.execute();
4 //update will refresh the dag and the lists
5 update();
6 }
```

Listing IV.10 – Exécution d'une commande d'assignation

La classe *ModelAccess* offre un accès à ces structures via les méthodes *getDag*, *getUndoList* et *getRedoList*. Celles-ci sont appelées principalement par les commandes lors de leurs exécutions.

Enfin, nous observons une vue d'ensemble de l'implémentation du modèle D.A.C à la Figure IV.8.

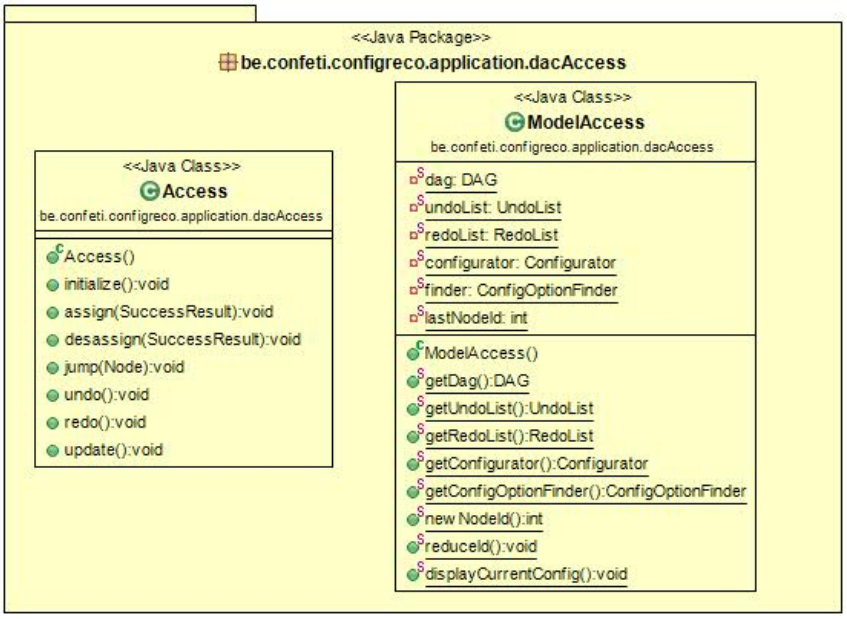


FIGURE IV.7 – Package "*dacAccess*"

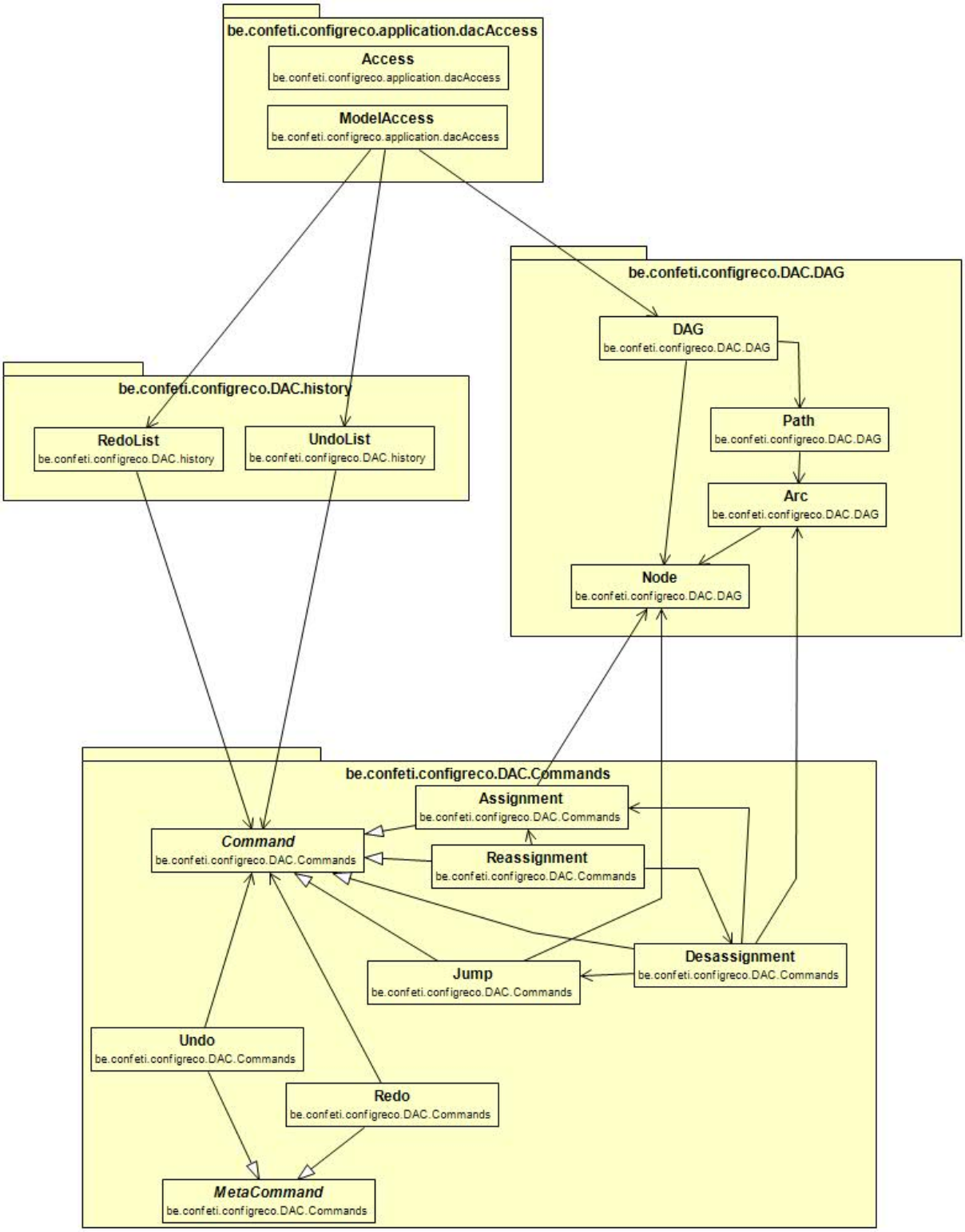


FIGURE IV.8 – Modèle D.A.C

Chapitre V

Étude de cas

Dans le but d'évaluer si notre modèle et notre implémentation résolvent bien les problèmes fixés, nous allons établir un scénario de configuration. Nous évaluerons les résultats obtenus en fonctions des bénéfices qu'offre le modèle D.A.C en comparaison avec d'autres modèles.

V.1 Scénario

Soit un configurateur proposant à un étudiant en informatique de choisir son programme de cours pour sa première année à l'université.

Le scénario est le suivant : l'étudiant est face au configurateur (comprenant l'implémentation du modèle D.A.C) et y trouve une liste de cours potentiels. Chacun des cours a un intitulé accompagné du nombre de crédits associés :

- Intro à la programmation - 11 crédits : True ou False
- Système d'informations - 3 crédits : True ou False
- Fonctions et concepts des ordinateurs - 7 crédits : True ou False
- Mathématiques générales - 6 crédits : True ou False
- Analyse mathématique - 5 crédits : dépend de Mathématiques générales
- Mathématiques pour l'informatique - 2 crédits
- Laboratoire de programmation - 3 crédits
- Faits et décision économiques - 6 crédits
- Approche anthropologique de l'informatique - 3 crédits
- Sciences religieuses - 3 crédits
- Langues - 3 crédits : Anglais ou Néerlandais

Pour chacun des cours, il doit indiquer s'il souhaite l'avoir en première année ou non (le cours est marqué *true* ou *false*), sauf pour l'option langue où il doit stipuler quelle langue il choisit (anglais ou néerlandais). Une contrainte est placée sur le cours "Analyse mathématique", celui-ci dépend du cours de "Mathématiques générales". Si l'étudiant choisi "Mathématiques générales", le cours "Analyse mathématique" sera automatiquement ajouté à son cursus. S'il décide de ne pas le prendre le cours de mathématiques, le cours d'analyse correspondant ne sera pas non plus sélectionné.
L'étudiant n'est pas certain des cours qu'il souhaite suivre et décide de tester plusieurs combinaisons.

1. Il commence par établir une première configuration en indiquant les cours qu'il souhaite suivre (Intro à la programmation, Système d'informations, Mathématiques générales, Laboratoire de programmation, Faits et décision économiques) ou non (Mathématiques pour l'informatique, Approche anthropologique de l'informatique et Sciences religieuses), il choisit également l'anglais comme option de langue.
2. Il change ensuite d'avis et décide de finalement prendre le cours "Mathématiques pour l'informatique", marqué non-désiré avant.
3. Après cela, il recommence une nouvelle configuration. Les deux premiers choix sont identiques à la configuration précédente mais il décide juste après d'inclure le cours "Fonctions et concepts des ordinateurs", qu'il avait ignoré auparavant. Cette fois-ci il ne désire pas prendre "Mathématiques générales".
4. Il continue sa progression en assignant les autres cours. Il termine par le néerlandais comme choix de langue.
5. N'étant pas satisfait par un de ses anciens choix, il effectue quatre UNDO, décide de finalement ne pas prendre le cours "Fait et décisions économiques" puis ré-exécute les assignations qui suivaient.
6. Enfin, il change une dernière fois d'avis et prend finalement le cours de "Mathématiques générales" puis termine sa configuration.

V.2 Application du modèle D.A.C

Voici la suite de commandes a exécuter pour résoudre le scénario grâce au modèle D.A.C. Chaque point correspond à une étape du scénario.

1. Il exécute la première suite de neuf assignations.
2. Il assigne ensuite "Mathématiques pour l'informatique" à *true*. Le cours étant marqué *false* au moment de l'assignation, cela provoque une réassignation. Celle-ci résulte en cette suite de commande : un saut (retour au nœud précédent l'assignation), le rejeu des assignations suivantes dans un nouveau chemin et enfin l'assignation de la variable à "false".
3. Pour recommencer une configuration depuis le début, il effectue un saut jusqu'au nœud racine. Les deux premiers choix sont de nouveau les mêmes que lors de sa précédente configuration. Ensuite il choisit un nouveau cours, précédemment resté non-assigné (Fonctions et concepts des ordinateurs). Cela provoque la création d'un nouveau chemin.
4. Il assigne sept cours.
5. Il exécute quatre UNDO, le faisant remonter dans le chemin. Il assigne ensuite le cours "Fait et décisions économiques" à une nouvelle valeur puis REDO les trois commandes qui suivaient l'assignation originelle.
6. Il décide de prendre le cours "Mathématiques générales", qu'il n'avait pas choisi lorsqu'il a recommencé sa configuration , créant une réassignation (un saut, supprimant la *redo list* + une assignation) et un nouveau chemin.

Nous observons dans la Figure V.2 l'*undo list* et la *redo list* dans leurs états à la fin de la configuration. Celles-ci ne contiennent pas toutes les commandes que l'utilisateur

a effectué. En effet, certaines ont été UNDO, se plaçant dans la *redo list*, cette dernière a ensuite été supprimée à plusieurs reprises, dû à l'exécution de plusieurs sauts. Pour rappel, voici le format des commandes présentes dans l'*undo list* :

- $a(v, val, o, t)$ ou $r(v, val, o, t)$: Assignation ou Réassignation où v est la variable, val est la valeur, o est le nœud d'origine, t est le nœud cible. La réassignation est accompagnée, dans la liste, d'une désassignation et d'une nouvelle assignation.
- Désassignation : $d(v, o, t)$ où v est la variable désassignée, o est le nœud d'origine et t est le nœud cible. La désassignation est suivie dans la liste d'un saut et éventuellement une liste d'assignations rejouées.
- Saut : $j(o, t)$ où o est le nœud d'origine et t le nœud cible.

Nous retrouvons donc dans la Figure V.2, l'*undo list* avec une première suite d'assignations, formant la première configuration. Celle-ci est directement suivie de la réassignation du cours "Mathématiques pour l'informatique", qui a provoqué la création d'un nouveau chemin. Puis l'exécution d'un premier saut jusqu'au nœud racine a permis à l'étudiant de commencer une troisième configuration à la fin de laquelle le cours "Mathématiques générales" a été placé à *false*. Il s'est ensuite ravisé et à réassigné le cours à *true*, créant un quatrième chemin et une quatrième configuration.
Le DAG correspondant à cette *undo list* est montré à la Figure V.1 dans laquelle nous avons placé uniquement la suite de nœuds créé (flèches pleines) et les sauts (flèches pointillées) afin de ne pas trop encombrer l'image. Les assignations effectuées peuvent être retrouvées aisément grâce à l'*undo list* qui indiquent les nœuds d'origines et cibles pour chacune d'entre elles.

Si l'étudiant effectue maintenant un saut sur le dernier nœud de chacun des chemins afin de comparer ses différents choix, il y trouvera les différentes configurations suivantes, correspondant bien à la séquence de commandes exécutées. Notons que dans ces configurations, nous observons la cours "Analyse Mathématique" assigné à la même valeur que le cours "Mathématiques générales". Cela prouve bien que la valeur de ce cours a été forcé par une propagations. Cela se confirme également lorsque nous observons l'*undo list* (Figure V.2) dans laquelle le cours d'analyse n'a jamais été assigné directement.

- La Figure V.3 représente la configuration associée au nœud 9.
- La Figure V.4 symbolise la configuration correspondant au nœud 15.
- La troisième configuration (nœud 27) est montrée à la Figure V.5.
- Enfin, la dernière configuration associée au nœud 34, le nœud courant est visible à la Figure V.6.

V.3 Évaluation

La principale force du modèle D.A.C est la combinaison du DAG et des history lists. Cela offre à l'utilisateur une vue claire de ses configurations grâce à un DAG uniquement orienté assignations, ce que n'aurait pas su fournir un modèle linéaire (tels que le RLU ou le triadic model). La présence du DAG et de la commande saut permettent également la comparaison de configurations de manière efficiente. Dans le scénario présenté, l'utilisateur peut, grâce au modèle D.A.C, effectuer des sauts en fin de configuration afin de comparer ses configurations, chose qu'aucun des modèles linéaires n'auraient permis de

faire.
Les history lists, orientées commandes, sont également présentes dans le modèle D.A.C, contrairement aux autres modèles possédant un graphe (US&R, history tree, UNDO sélectif direct et d'UNDO sélectif en cascade). Ces listes permettent à l'utilisateur d'avoir un historique des commandes en vigueur correspondant à la structure du DAG. Chaque commande présente dans la liste indique les noeuds origine et cible, permettant à l'utilisateur de savoir quand telle assignation a été effectuée et où elle se situe dans le DAG.
Le fait qu'UNDO une commande de l'*undo list* produise un effet sur le DAG en supprimant les noeuds et arcs ciblés permet d'éviter un DAG trop encombré, comme il pourrait l'être avec un modèle comme l'UNDO sélectif direct dont l'UNDO ajoute des noeuds à la structure. Enfin, l'UNDO linéaire associé à un REDO sélectif est une fonctionnalité que l'on ne retrouve pas dans les précédents modèles. Le REDO sélectif offre une certaine liberté quant à la réexécution des commandes de la *redo list*. Celle-ci étant régulièrement vidée, nous ne risquons pas non plus d'obtenir une liste trop grande comme cela pourrait arriver dans les autres modèles linéaires mis à part le RLU. En revanche, ce dernier n'aurait pas permis de recommencer une configuration depuis le début, comme nous l'avons fait dans le scénario, sans perdre la configuration existante. En effet il aurait été obligé d'exécuter des UNDO jusqu'à revenir à l'origine puis l'exécution d'une nouvelle assignation aurait vidée la *redo list*. Les modèles de l'history tree et de l'US&R auraient sauvegardé la configuration et créé une nouvelle branche mais l'étudiant aurait du voyager par méta-commandes pour revenir ensuite à la configuration originale, au lieu d'utiliser un saut.
En ce qui concerne l'undo sélectif direct, le système d'UNDO qui y est associé aurait ajouté des noeuds au graphe représentant les commandes inverses de celles UNDO, encombrant inutilement le graphe. Le graphe des modèles d'UNDO sélectif (direct et en cascade) n'est pas orienté assignations comme celui du modèle D.A.C mais est orienté commandes. Cela veut dire que si une réassignation aurait été effecuté dans un de ces modèles, elle aurait été ajoutée comme une commande à la suite du chemin courant sans créer de nouvelle branche contenant la nouvelle configuration. Pour que cette nouvelle branche se crée, l'utilisateur devrait sélectionner manuellement le noeud précédant l'assignation originale et ensuite exécuter une nouvelle commande (et potentiellement rejouer les assignations qui suivaient en utilisant des REDO). Dans notre modèle, ce procédé de réassignation est automatisé.

Nous constatons donc que le modèle D.A.C convient à la résolution du scénario et permet à l'utilisateur une navigation plus efficace et efficiente que les autres modèles présentés. De cette navigation découle la possibilité pour l'utilisateur de comparer sans efforts les différentes configurations créées.

```
UNDO LIST
*********

a(Intro à la programmation (11),true,0,1)
a(Système d'informations (3),true,1,2)
a(Mathématiques générales (6),true,2,3)
a(Mathématiques pour l'informatique (2),false,3,4)
a(Laboratoire de programmation (3),true,4,5)
a(Faits et décisions économiques (6),true,5,6)
a(Approche anthropologique de l'informatique (3),false,6,7)
a(Sciences religieuses (3),false,7,8)
a(Langue,Anglais (3),8,9)
r(Mathématiques pour l'informatique (2),true,9,15)
        => d(Mathématiques pour l'informatique (2),9,14)
        > j(9,3)
        >a(Laboratoire de programmation (3),true,3,10)
        >a(Faits et décisions économiques (6),true,10,11)
        >a(Approche anthropologique de l'informatique (3),false,11,12)
        >a(Sciences religieuses (3),false,12,13)
        >a(Langue,Anglais (3),13,14)
        => a(Mathématiques pour l'informatique (2),true,14,15)

j(15,0)
a(Intro à la programmation (11),true,0,1)
a(Système d'informations (3),true,1,2)
a(Fonctions et concepts des ordinateurs (7),true,2,16)
a(Mathématiques générales (6),false,16,17)
a(Mathématiques pour l'informatique (2),false,17,18)
a(Laboratoire de programmation (3),true,18,19)
a(Faits et décisions économiques (6),false,19,24)
a(Approche anthropologique de l'informatique (3),true,24,25)
a(Sciences religieuses (3),true,25,26)
a(Langue,Néérlandais (3),26,27)
r(Mathématiques générales (6),true,27,34)
        => d(Mathématiques générales (6),27,33)
        > j(27,16)
        >a(Mathématiques pour l'informatique (2),false,16,28)
        >a(Laboratoire de programmation (3),true,28,29)
        >a(Faits et décisions économiques (6),false,29,30)
        >a(Approche anthropologique de l'informatique (3),true,30,31)
        >a(Sciences religieuses (3),true,31,32)
        >a(Langue,Néérlandais (3),32,33)
        => a(Mathématiques générales (6),true,33,34)


REDO LIST
*********
```

FIGURE V.1 – History lists résultant de la configuration de l'étudiant

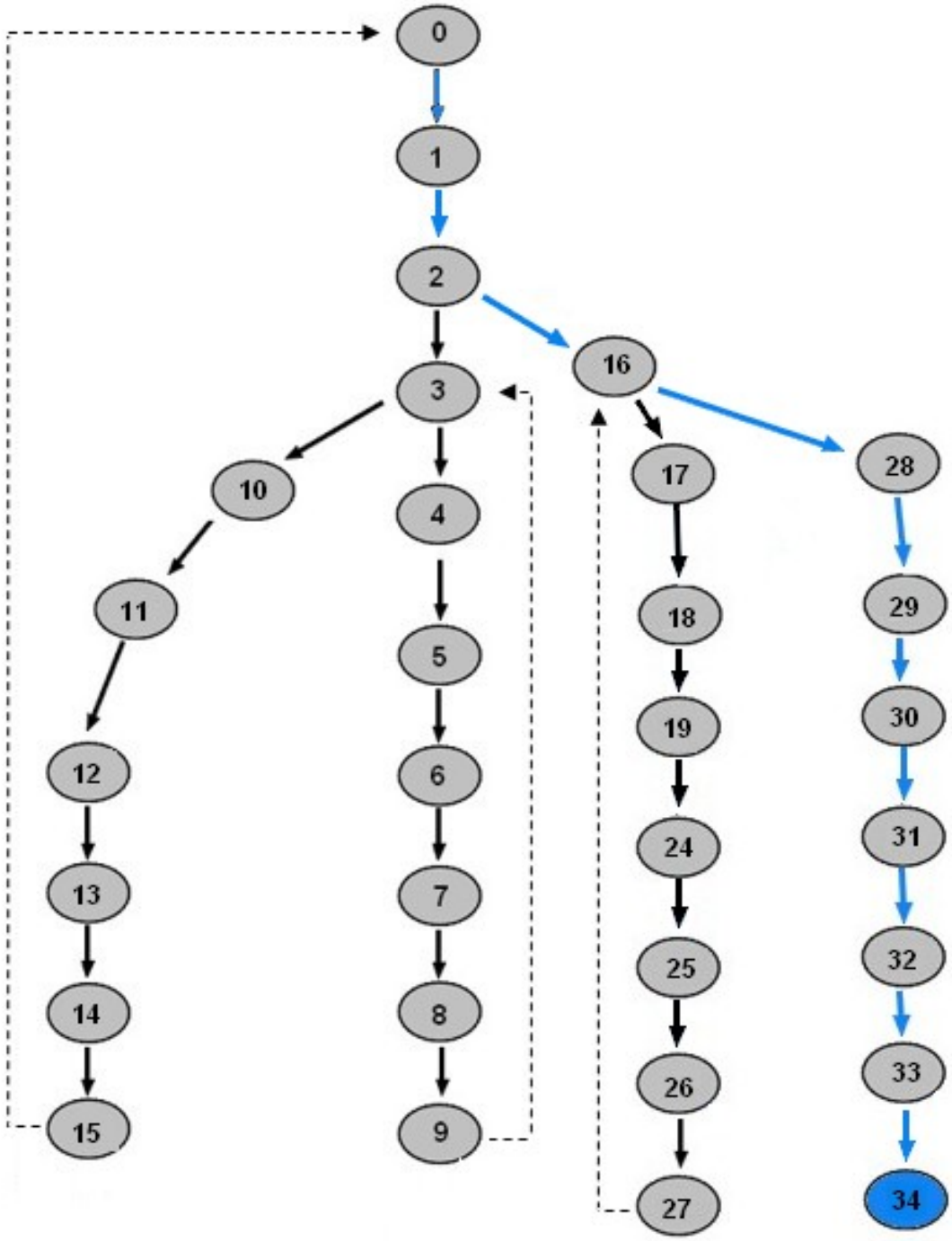


FIGURE V.2 – DAG résultant de la configuration de l'étudiant

```
ConfigAssignment{{Intro à la programmation (11)}, value=true}
ConfigAssignment{{Système d'informations (3)}, value=true}
ConfigAssignment{{Mathématiques générales (6)}, value=true}
ConfigAssignment{{Analyse mathématique (5)}, value=false, message=forced}
ConfigAssignment{{Mathématiques pour l'informatique (2)}, value=false}
ConfigAssignment{{Laboratoire de programmation (3)}, value=true}
ConfigAssignment{{Faits et décisions économiques (6)}, value=true}
ConfigAssignment{{Approche anthropologique de l'informatique (3)}, value=false}
ConfigAssignment{{Sciences religieuses (3)}, value=false}
ConfigAssignment{{Langue, 10}, value=Anglais (3)}
```

FIGURE V.3 – Première configuration complète

```
ConfigAssignment{{Intro à la programmation (11)}, value=true}
ConfigAssignment{{Système d'informations (3)}, value=true}
ConfigAssignment{{Mathématiques générales (6)}, value=true}
ConfigAssignment{{Analyse mathématique (5)}, value=true, message=forced}
ConfigAssignment{{Mathématiques pour l'informatique (2)}, value=true}
ConfigAssignment{{Laboratoire de programmation (3)}, value=true}
ConfigAssignment{{Faits et décisions économiques (6)}, value=true}
ConfigAssignment{{Approche anthropologique de l'informatique (3)}, value=false}
ConfigAssignment{{Sciences religieuses (3)}, value=false}
ConfigAssignment{{Langue}, value=Anglais (3)}
```

FIGURE V.4 – Seconde configuration complète

```
ConfigAssignment{{Intro à la programmation (11)}, value=true}
ConfigAssignment{{Système d'informations (3)}, value=true}
ConfigAssignment{{Fonctions et concepts des ordinateurs (7)}, value=true}
ConfigAssignment{{Mathématiques générales (6)}, value=false}
ConfigAssignment{{Analyse mathématique (5)}, value=false, message=forced}
ConfigAssignment{{Mathématiques pour l'informatique (2)}, value=false}
ConfigAssignment{{Laboratoire de programmation (3)}, value=true}
ConfigAssignment{{Faits et décisions économiques (6)}, value=false}
ConfigAssignment{{Approche anthropologique de l'informatique (3), value=true}
ConfigAssignment{{Sciences religieuses (3)}, value=true}
ConfigAssignment{{Langue}, value=Néérlandais (3)}
```

FIGURE V.5 – Troisième configuration complète

```
ConfigAssignment{{Intro à la programmation (11)}, value=true}
ConfigAssignment{{Système d'informations (3)}, value=true}
ConfigAssignment{{Fonctions et concepts des ordinateurs (7)}, value=true}
ConfigAssignment{{Mathématiques générales (6)}, value=true}
ConfigAssignment{{Analyse mathématique (5)}, value=true}
ConfigAssignment{{Mathématiques pour l'informatique (2)}, value=false}
ConfigAssignment{{Laboratoire de programmation (3)}, value=true}
ConfigAssignment{{Faits et décisions économiques (6)}, value=false}
ConfigAssignment{{Approche anthropologique de l'informatique (3),value=true}
ConfigAssignment{{Sciences religieuses (3)}, value=true}
ConfigAssignment{{Langue}, value=Néérlandais (3)}
```

FIGURE V.6 – Dernière configuration complète

Conclusion

Le but de ce mémoire était de concevoir un système de navigation dans un cadre de configuration afin de permettre à l'utilisateur de revenir en arrière et de comparer ses différentes réalisations. Nous pouvons affirmer que cet objectif est atteint grâce à la réalisation du modèle D.A.C. Avant de développer de modèle, nous nous sommes d'abord concentrés sur la partie du retour-arrière ou UNDO. Nous avons recherché et étudié les différents modèles d'UNDO existants afin de savoir lesquels pouvaient convenir à un tel système ou quels bases ils pouvaient nous apporter pour en établir un nouveau.

Après avoir fait des recherches sur ces modèles, nous avons établi des critères qui devaient être respecté afin de pouvoir convenir à la configuration. Nous avons conclu que certains critères étaient respectés par des modèles partageant les mêmes caractéristiques. Nous avons donc repris ces avantages et les avons combinés dans un nouveau modèle, plus efficient et conçu pour la configuration. Ces avantages sont : la présence d'un graphe des états du programme qui deviendra dans notre cas, un graphe de configurations et un système de sélection direct des noeuds le composant. En effet, nous avons constaté que ces aspects augmentait l'efficience de la navigation.

Nous avons ensuite présentés les différents aspects de la configuration à prendre en compte dans le nouveau modèle que nous avons développé. Celui-ci, appelé D.A.C (DAG-Assignment-Command), reprend les notions de graphe orienté acyclique (DAG) avec sélection directe et de liste de commandes. Le DAG contient exclusivement des assignations amenant à des configurations. Il est construit grâce aux commandes exécutées par l'utilisateur, qui sont rapportées dans les history list. La validation de chaque assignation est vérifiée par le moteur de configuration faisant partie de la suite d'outils CONFETOOLS créé par la start-up CONF&TI.

Nous avons ensuite implémenter ce modèle en utilisant une API tirée de CONFETOOLS. Celle-ci fournit des services pour utiliser le configurateur. Nous les avons donc utilisé afin qu'il y ait une interaction avec le modèle D.A.C lors d'assignation ou de désassignation de variables dans le configurateur. Lors de chaque commande de l'utilisateur, un effet se produit sur le DAG ainsi que sur les history list. Nous avons ensuite appliqué ce modèle à un scénario de configuration pour prouver qu'il s'appliquait bien dans un cadre de configuration comme attendu et qu'il résolvait des problèmes qu'auraient rencontrés d'autres modèles.

Cependant, certains aspects du modèle D.A.C peuvent être améliorés. En effet, dû à des contraintes de temps, certains problèmes n'ont pas pu être réglées. Tout d'abord, la

gestion des chemins dans le DAG n'est pas optimale car elle est redondante. En effet, un objet *Path* représentant un chemin n'est pas utile dans l'implémentation car nous pouvons inférer un chemin depuis une liste d'arcs. Cela provoque une redondance et peut potentiellement consommer beaucoup de mémoire lorsque nous sommes face à un vaste espace de possibilité.
La gestion des *id* des noeuds est faite, dans l'implémentation actuelle, de manière incrémentale. Ce système est à revoir, peut-être grâce à une génération d'*id* unique sur base du contenu de la configuration.
Enfin, de nouveau pour des contraintes de temps, seule la partie back-end du modèle a été développée. Nous pourrions envisager un futur travail reposant sur l'amélioration du modèle et le développement d'une partie front-end intégrée dans le configurateur. Pour la représentation du DAG nous pourrions nous baser sur les différentes représentations contenues dans le chapitre III (par exemple la Figure III.1). Il faudrait faire en sorte que chaque clic sur un noeud provoque un saut. Il faudrait également rendre chaque commande la *redo list* sélectionnable directement.

Bibliographie

[1] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact.*, 1(3) :269–294, September 1994.

[2] James E. Archer, Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst.*, 6(1) :1–19, January 1984.

[3] Karel Jakubec, Marek Polák, Martin Necaský, and Irena Holubová. Undo/redo operations in complex environments. *Procedia Computer Science*, 32(0) :561 – 570, 2014. The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014).

[4] Aaron G. Cass, Chris S. T. Fernandes, and Andrew Polidore. An empirical evaluation of undo mechanisms. In *Proceedings of the 4th Nordic Conference on Human-computer Interaction : Changing Roles*, NordiCHI '06, pages 19–27, New York, NY, USA, 2006. ACM.

[5] Germain Saval. CONFETOOLS : A tool to generate an inter- active configuration interface and runtime. In Raul Mazo Pena Gilles Perrouin, Mathieu Acher, editor, *6ème Journée Lignes de Produits*, pages LDPIDM–4, Paris, France, November 2013.

[6] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing tvl, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS ?10), Linz, Austria, January*, pages 27–29, 2010.

[7] Jeffrey Scott Vitter. Us&r : A new framework for redoing (extended abstract). *SIGSOFT Softw. Eng. Notes*, 9(3) :168–176, April 1984.

[8] Aaron G. Cass and Chris S. T. Fern. Modeling dependencies for cascading selective undo. In *In IFIP INTERACT 2005 Workshop on Integrating Software Engineering and Usability Engineering*, 2005.

[9] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling : Syntax and semantics of tvl. *Science of Computer Programming*, 76(12) :1130–1143, 2011.

Annexes

Annexe A

```
root Rexel {
   group allof {
      Logement {
      int surface;
      Logement.surface >= 35 -> count(Circuit) >= 2;
      group allof {
         Chambre [0..5] ,
         Cuisine [0..5] ,
         Sejour [0..5] ,
          WC [0..5] ,
          Coffret {
             group someof {
                Porte ,
                PeigneHorizontal [1..5] {
                   group allof {
                      Protection {
                         group [0..2] {
                            InterDiff ,
                            Parafoudre
                         }
                      },
                      CircuitProtege [1..5] {
                         group allof {
                            ProtectionCircuit {
                               group oneof {
                                  Disjoncteur ,
                                  DisjDiff ,
                                  Fusible
                               }
                            },
                            Circuit [1..5] {
                               group oneof {
                                  CircuitEclairage ,
                                  CircuitPC16A ,
                                  CircuitSpecialise
}}}}}}}}}}}
```

Listing 1 – Exemple de document TVL représentant un réseau de distribution d'électricité [5]